StreamJoin: A Generic Database Approach to Support the Class of Stream-Oriented Applications

Clara Nippl
Department of Computer Science
Technical University of Munich
D-80290 Munich, Germany
nippl@in.tum.de

Ralf Rantzau, Bernhard Mitschang
Institute of Parallel and Distributed High-Performance Systems, University of Stuttgart
D-70565 Stuttgart, Germany
{rantzau, mitsch}@informatik.uni-stuttgart.de

Abstract

Today many applications routinely generate large quantities of data. The data often takes the form of (time) series, or more generally streams, i.e. an ordered sequence of records. Analysis of this data requires stream processing techniques which differ in significant ways from what current database analysis and query techniques have been optimized for. In this paper we present a new operator, called StreamJoin, that can efficiently be used to solve stream-related problems of various applications, such as universal quantification, pattern recognition and data mining. Contrary to other approaches, StreamJoin processing provides rapid response times, a non-blocking execution as well as economical resource utilization. Adaptability to different application scenarios is realized by means of parameters. In addition, the StreamJoin operator can be efficiently embedded into the database engine, thus implicitly using the optimization and parallelization capabilities for the benefit of the application. The paper focuses on the applicability of StreamJoin to integrate application semantics into the DBMS.

1. Introduction

All-quantification can be seen as a primitive in a number of upcoming data analysis scenarios as e.g. time series analysis in finance, genomic sequence matching in biochemistry, and frequent itemset discovery in databases. Since these application areas show a tremendous growth in data volume and because the data analysis problems are getting more complex, an efficient processing of the all-quantification close to the data is getting vital to the success of these applications.

Our approach is twofold: Firstly, we define a new operator, called StreamJoin that directly evaluates all-quantification in an efficient manner. Secondly, we show how this operator can efficiently be integrated into a state-of-the-art DBMS.

Please note that the implementation strategy as well as a detailed description of the StreamJoin algorithm are covered in another paper [17]. In this paper we concentrate on identifying the usage of the StreamJoin primitive within the applications mentioned below:

- *Universal Quantification*: Forthcoming applications stress the need for an efficient implementation of universal quantification concepts. Given e.g. a decision support system (DSS), a frequently formulated query type is the following: "Find the customers/suppliers/stores that buy/supply/sell *all* items that satisfy a particular condition."
- *Sequences*: Recently, there is a growing interest in periodicity search [10] in time-related databases. For instance, databases for stock analysis often process queries like: "Find *all* stocks that monotonically fall/rise in a given period of time."
- *Pattern Recognition*: In the domain of molecular biology, scientists frequently attempt to match functionally unknown proteins against a protein coding database of known proteins. If a match is found, it is likely that the proteins are functionally related. Thus, given a sequence and a protein coding database, the problem can be formulated as follows: "Find the item sequences in the database that contain *all* items of the pattern sequence in the given order."
- *Data Mining*: One of the core mechanisms of many data mining algorithms [1] is a phase that evaluates patterns called frequent itemsets. A frequent itemset is a set of items appearing together in a number of database records meeting a user-defined threshold, called minimum support. The final itemsets are usually derived using a set of candidate itemsets. That means, a candidate itemset is established as being a frequent itemset if the number of transactions containing *all* items in the candidate itemset exceeds the predefined support.

- *Digital Libraries*: Modern digital libraries offer a profiling service to keep track of their users' individual reading interests. A profile usually consists of a set of keywords. One task of the profiling service is to find the documents that contain *all* keywords. In some cases, the words in the documents have to fulfill certain additional position requirements like a specific order [15].

If we analyze in more detail the data involved in these query types, we recognize that it often takes the form of *streams*, i.e. a *variable* number of (intermediate result) *tuples* that share a common feature. A stream corresponds for instance to the set of transactions that contain a specific item or the set of documents that contain a specific keyword. The subsequent processing within an application domain refers to certain relationships *in-between* the streams. For instance, the final result of the universal quantification is given by those tuples that are included in all streams. In the case of profile evaluation, the tuples in subsequent streams have to fulfill the required positioning requirements.

Given the above, we conclude that there is an important class of applications where there is a need to perform *stream analysis* instead of traditional data analysis, achieved for instance through aggregations. A further common feature of all these applications is the fact that the number of *streams* is also *variable* and not known a priori, as it corresponds e.g. to the number of items in an itemset, or the number of search terms in a profile.

In this paper, we introduce a new database operator, called *StreamJoin*, that can efficiently solve the problems mentioned above. Thereby, we concentrate on the applicability of this strategy for various scenarios covering universal quantification, sequences, and pattern recognition. In our previous work, we have already employed the StreamJoin operator to data mining tasks [18] and profile evaluation [15].

Consequently, the paper is organized as follows. In Section 2 we describe the basic functionality of the StreamJoin operator. Section 3 demonstrates the applicability of this technique for universal quantification. Section 4 and 5 exemplify the usage of StreamJoin for sequence analysis, respectively pattern recognition in genomic databases. A performance evaluation is given in Section 6. Finally, Section 7 presents some concluding remarks.

2. The StreamJoin Operator

The arity of the StreamJoin operator is 1, contrary to traditional join operators. Thus, it can also be regarded as a specific aggregation or restriction operator. Two distinguished attributes divide the entire input into *groups* that in turn are subdivided into *streams*. These attributes are handed over to the StreamJoin operator as parameters, called *GroupId* and *StreamId*. From the StreamJoin's

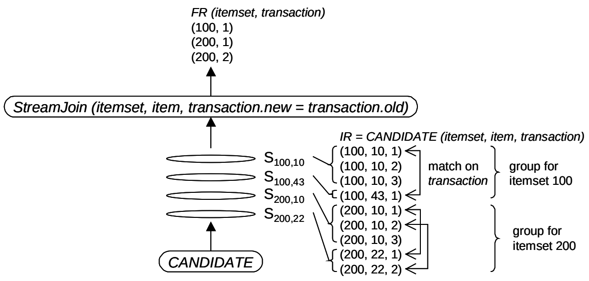

Figure 1: StreamJoin processing for an input stream of candidate itemsets.

point of view any of the input attributes can act as *GroupId* or *StreamId*, provided that the input is grouped on these attributes.

The main functionality of this operator is to join subsequent streams of a given group. Thereby, both the join attributes as well as the corresponding join conditions are given as parameters. Obviously, the join attributes, or *JoinIds*, are different from the *GroupId* or *StreamId* attributes.

Hence, we define the following signature for the StreamJoin operator: *StreamJoin (GroupId, StreamId, predicate(JoinId1, JoinId2, ...))*

The first two parameters specify the attributes that define a group and the streams within a group. The subsequent parameter defines the join predicate as a boolean expression over the join condition. This join predicate defines condition(s) between attribute values of the current stream and those of the previous stream. This is expressed by the suffixes *new* and *old*. Thus, *JoinId.new* represents the value of the attribute *JoinId* in the current stream and *JoinId.old* represents the value of the same attribute in the previous stream.

For a better understanding, we illustrate this functionality by using the following example. Consider the data mining problem of finding frequent itemsets. Assume that some kind of pre-processing has delivered a set of candidate itemsets organized in a table *CANDIDATE (itemset, item, transaction)* as depicted in Fig. 1. We are interested in those transactions that contain *all* items of any given itemset.

In order to provide this result, we simply join subsequent streams on the *transaction* attribute. Thereby, an instance of the StreamJoin operator that performs this functionality has the signature *StreamJoin (itemset, item, transaction.new = transaction.old)*. We assume that the tuples of *CANDIDATE* are grouped on *itemset* and *item*. We call this input of the StreamJoin operator an intermediate result (*IR*) since it is usually the output of a more complex query execution plan (QEP) and not a base table as in this example.

The processing is illustrated in Fig. 1 for two example groups, 100 and 200. The group, i.e. candidate itemset, 100 is constituted of two streams, called $S_{100,10}$ and $S_{100,43}$. These streams define the transactions that contain item 10 and 43, respectively. The StreamJoin operator joins all streams of each group on the attribute *transaction*. This yields one tuple (100, 1) for the group 100 indicating that the transaction 1 contains all items of the candidate itemset 100, namely 10 an 43. The same join processing for the group 200 results in two tuples, namely (200,1) and (200, 2). Here, both transaction 1 and transaction 2 contain all items of the candidate itemset 200. Hence, the final result *FR* contains three tuples.

The implementation of StreamJoin is based on separate iterations, corresponding to the streams within a group. Thereby, hash-based data structures are used to memorize those tuples of the input that are candidates for the final result. In the first iteration, the number of candidates is equal to the number of tuples in the first stream. However, this number decreases in each iteration, since only those tuples are retained that satisfy the join condition. This process continues until the next group is reached. The tuples that survive all iterations satisfy the join condition for every stream of a group and are added to the output stream, e.g. the final result *FR* in Fig. 1. Please note that the number of tuples to be kept in memory at a given time is at most the size of a single stream. In addition, the intermediate result sizes decrease with each iteration. Thus, this strategy yields an economical memory consumption.

As already mentioned, the input of the StreamJoin operator has to be grouped on the *GroupId* and *StreamId* attributes. Obviously, this requirement can always be fulfilled by adequate sorting techniques. However, as shown later on in this paper, sorting can mostly be avoided. This results from the fact that often the necessary grouping for the StreamJoin operator implicitly comes via pre-processing steps performed earlier in the query execution plan.

Please note that this description of StreamJoin shows certain similarities to the evaluation of recursive queries in database systems [5]. Indeed, though the two areas seem radically different because of the approach and formalism used, they have some common features. First, the presented hash-based implementation of StreamJoin is similar to the transitive closure algorithm described in [11]. Second, both approaches apply a variable number of consecutive iterations and a stop condition to obtain the final result. Moreover, in both cases the number of iterations is not known a priori, being dependent on the value distribution of the input. However, the difference between the two approaches lies in the characteristics of the iterations. As already mentioned, StreamJoin processes its input *stream-wise* or *linearly*, i.e. a given input tuple is generally considered only once. In contrast, the evaluation of transitive closure, or recursion in general, requires a repeated processing of the input. This *cyclic* method may consider the same input tuple several times in order to produce the complete result. Another important difference between the two approaches lies in the fact that the StreamJoin processing reduces the intermediate result with each iteration, as the tuples which do not match the join predicate are eliminated. In contrast, transitive closure produces in each iteration new tuples that are added to the final result. Finally, there is also a major difference between the two stop conditions applied. The transitive closure algorithm stops when no new tuples are produced, i.e. the (intermediate) result set of a given iteration is identical to the one of the previous iteration. In contrast, the StreamJoin algorithm joins streams of the same group until the subsequent group is reached. Thus, the stop condition for joining is the fact that the value of the current tuple's *GroupId* attribute is different from that of the previous input tuple.

A thorough discussion of frequent itemset discovery involving StreamJoin and a more detailed description of the algorithm and implementation can be found in [17]. We introduced the StreamJoin operator only as far as is necessary for the understanding of this paper.

3. Universal Quantification

Complex queries containing quantifiers, also called quantified queries, become increasingly important in forthcoming applications, such as OLAP systems. However, relational database systems do not adequately support such queries. Effective support is needed both at the language level and in the underlying query processing system. As far as the first issue is concerned, quantified queries are usually expressed in SQL by various clauses like GROUP BY and counting, as well as predicates like ALL, ANY/SOME, (NOT) EXISTS, and (NOT) IN.

We first consider the example given in [8], a university database with two relations, *COURSE (courseNo, title)* and *TRANSCRIPT (studentId, courseNo, ...)*. The goal is to find the students who have taken all courses offered by the university. Two examples of expressing this query in SQL are presented below.

```
SELECT DISTINCT t1.studentId
FROM TRANSCRIPT t1
WHERE NOT EXISTS (
  SELECT * FROM COURSE c
  WHERE NOT EXISTS (
    SELECT * FROM TRANSCRIPT t2
    WHERE t2.studentId = t1.studentId
    AND t2.courseNo = c.courseNo))
```

```
SELECT t.studentId
FROM TRANSCRIPT t
GROUP BY t.studentId
HAVING COUNT(t.courseNo) = (
  SELECT COUNT(courseNo)
  FROM COURSE)
```

Obviously, these formulations are not intuitive and in addition difficult to optimize. Language extensions have already been proposed in the literature [12] [19] and meanwhile considered as additional predicates in the SQL:1999 standardization [9]. In this paper, we concentrate on the support provided by relational query processors.

Since all-quantification can be seen as a division operation, there is a direct relationship from StreamJoin to a division operation. In fact, StreamJoin can mimic a division operation by simply using an equi-join predicate involving the division column.

3.1. Related Work

[6] presents a comprehensive treatment of universal quantification from the query level to evaluation. According to this analysis, plans implementing the all-quantification with an anti-semijoin are superior to all other alternatives. However, this approach is best supported in object-oriented and object-relational models. Thus, it is still an open problem how to deal with universal quantifiers in data warehouse applications, that are mostly based on a relational star or snowflake schema. Because of high data volumes, especially in these environments effective support is needed, and data reorganization has to be avoided.

Universal quantification is evaluated in [8] by a hash-based division algorithm. However, this approach applies only for a special class of queries, namely those for which the quantifier's range constitutes a closed formula [6]. In [7] generalized join and aggregation operators are presented. However, the scope of the paper is restricted only to traditional aggregation operations over groups, such as *average, max, count* etc., and is not applicable for all-quantification. [19] proposes a generalized quantifier framework that defines a completely new query subsystem. Thus, it requires significant changes within the query execution system, since special indexes and multi-dimensional structures have to be built for all relations. Moreover, the results are not directly applicable for large-scale applications, such as e.g. OLAP. In addition, web technology can benefit from universal quantification as expressed for example in the web join approach like in [3].
In the following, in order to assess the applicability of the different implementation approaches, namely hash-join, anti-semijoin and StreamJoin, we will consider two example scenarios employing universal quantification.

3.2. Example Scenario 1: The University Database

We first consider the university database example mentioned above. Again, the request is to find the students who have taken all courses offered by the university. As already mentioned, this database contains two relations, *COURSE (courseno, title)* and *TRANSCRIPT (studentid, courseno, ...)*.

The Hash-Division Algorithm. In [8] the *TRANSCRIPT* relation is called the dividend, the *COURSE* table the divisor and the division result is called quotient. The hash-division algorithm uses two hash tables, one for the divisor and one for the quotient. For each tuple in the quotient table a bitmap is kept with one bit for each divisor tuple. First, all divisor tuples are inserted into the divisor table. Next, the algorithm consumes the dividend relation. If a matching tuple is found in the divisor table, the dividend tuple is newly inserted as a candidate into the quotient table and its bitmap is initialized with zeros except for the bit corresponding to the matching divisor tuple. However, if it is already present, the only thing to be done is to modify the associated bitmap by turning the bit corresponding to the matching divisor tuple to 1. When all tuples of the dividend relation are consumed, the quotient consists of those tuples in the quotient table for which the corresponding bitmap contains no zeros.

The Anti-Semijoin Approach. As described in [6], in an object-oriented model the *N:M* relationship *enrolled* between students and courses is typically modeled by a set-valued attribute *enrolledCourses* for each student. Thus the all-quantification can be resolved by an anti-semijoin on the two tables *TRANSCRIPT* and *COURSE* using the condition $courseNo \notin enrolledCourses$. The anti-semijoin adds to the output stream only those tuples, i.e. students, for which no join partner has been found. This is equivalent to the fact that all courses are included in the *enrolledCourses* attribute of the given student item, hence the student has attended all courses.
However, in a relational schema this approach cannot be applied. Obviously, the anti-semijoin on *TRANSCRIPT* and *COURSE*, using the condition $courseNo \neq courseNo$, yields as a result an empty set if referential integrity is guaranteed. Thus, the anti-semijoin strategy cannot be applied for the evaluation of universal quantification in e.g. data warehouses that adopt a relational star or snowflake schema.

The StreamJoin Approach. Fig. 2 presents the QEP for the evaluation of the all-quantification via the StreamJoin operator. By joining the two tables *TRANSCRIPT* and *COURSE* using e.g. an index on *courseNo* for the *TRANSCRIPT* table, the intermediate result *IR* consists of separate streams for each course, containing the students that have taken that course. These streams are called $S_{Databases}$, $S_{Compilers}$ and $S_{Graphics}$ in our example. In order to obtain the students that have participated in all courses, a join of these streams on the *studentId* attribute is necessary. This operation is performed by the StreamJoin operator, yielding in the example from Fig. 2 one tuple for the final result *FR*. Please note that the parameter corresponding to the *GroupId* is set to 0 (or any arbitrary constant value), as there is only one divisor set to be tested.

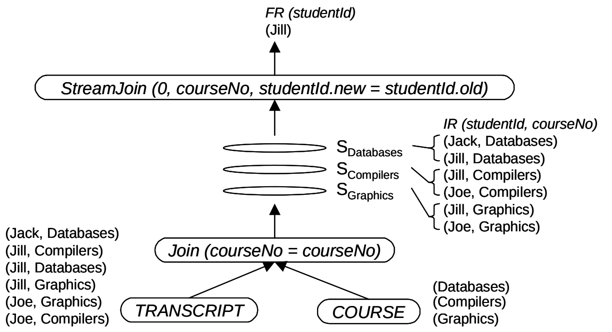

Figure 2: Evaluation of universal quantification with the Streamjoin approach.

As compared to the hash-division algorithm, this approach needs less buffer space, as the maximum memory consumption corresponds to the size of the first stream (in this example the two tuples of the stream $S_{Databases}$). As shown in Section 2, the intermediate result sizes decrease with each iteration because the tuples which do not match the join condition are eliminated. In contrast, the hash-division algorithm has to keep the whole divisor, i.e. the *COURSE* table (3 tuples), in memory, together with all quotient candidates and their corresponding bitmaps (3 tuples + 3 bitmaps). For this small example, this sums up to 6 tuples + 3 bitmaps that have to be kept permanently in memory for the hash-division algorithm, compared to at most 2 tuples, i.e. the number of distinct items in the first stream, that are kept in memory for the StreamJoin approach.

3.3. Example Scenario 2: Data Warehouse

Consider a data warehouse with a central *FACT* table, describing the sales in a given time period, and several dimension tables, e.g. *NATION* being one of them:

FACT (partkey, suppkey, nationkey, ...)
NATION (regionkey, nationkey, nationname, ...)
...

Assume that for marketing purposes, a user is interested in the query: "Find the suppliers who supply a part that is being sold in all nations of a region."
In the following, we analyze how this query can be solved by the various approaches.

The Hash-Division Algorithm. In this case, the divisor table is *NATION*, or, more precisely, it is constituted of several parts of this table, each part corresponding to a region. It results already from this aspect that the hash-division algorithm cannot be applied in a straightforward way, as it has to be extended to keep separate bitmaps for every divisor, i.e. every region. In addition, each *<partkey, suppkey>* combination has to be considered as a possible quotient candidate. Thus, the quotient table is constituted of all such combinations, each having in addition several bitmaps corresponding to the different regions. These memory requirements also show that the algorithm is not competitive for this type of query.

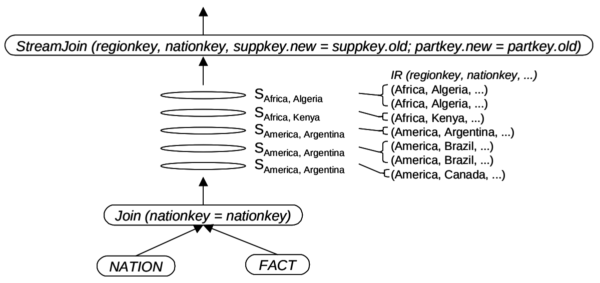

Figure 3: Simplified QEP for an OLAP query involving a StreamJoin operator.

The Anti-Semijoin Approach. Similar to Section 3.2.2, the approach is not applicable for this relational schema, since the anti-semijoin between the two tables on the *nationkey* attributes yields an empty set.

The StreamJoin Approach. Since *nationkey* is one of the dimensions of the central *FACT* table, suppose for simplicity purposes that there exists an index on this attribute. We further assume that the *NATION* table is sorted on the *regionkey* attribute. Please note that if this condition is not fulfilled by the physical databases design, it can be accomplished e.g. by a corresponding sort operator. By joining the *NATION* and *FACT* table, each region defines a group, containing the parts that have been sold in that region. In the example in Fig. 3 the groups are defined by the values *Africa* and *America*. Each group contains several streams, corresponding to the different countries of that region. For instance, the group *Africa* is constituted of the streams *Algeria* and *Kenya*. This intermediate result constitutes the input for the StreamJoin operator. Thus, the *GroupId* parameter of the StreamJoin operator as defined in Section 2 corresponds to the *regionkey* attribute, while the *StreamId* is set to *nationkey*. If we had joined the streams on the *suppkey* attribute alone, we would have obtained the suppliers whose different parts are sold in all countries of a region. However, the query contains as an additional constraint that it must be the same part that is sold in all countries. Thus, the join is defined on two attributes, namely *suppkey* and *partkey*. For both attributes, the join condition is equality, expressed by *suppkey.old = suppkey.new*, respectively *partkey.old = partkey.new*.

4. Sequences

Recently, patterns and sequences, especially time sequences, appear in various application domains. Typical examples are scientific experiments such as temperature records generated by sensors, business applications such as stock price indexes or bank account histories and medical applications such as cardiology data. Sequence processing is a challenging task for data mining purposes as well [24]. The corresponding (temporal) databases tend to be voluminous, thus forcing efficient algorithms to reduce processing overhead, such as communication costs etc. However, related work treats sequence analysis mostly on top of a database. In contrast, the StreamJoin operator processes sequences in an integrated fashion internal to the DBMS. This will be demonstrated in the following using as an example an application operating on financial data.

One interesting scenario in finance is to identify pairs of stocks whose prices track one another, i.e. show similar pricing over a specified period of time [14]. Suppose a database for stock analysis that contains the table *STOCKINFO (week, day, stockkey, price)*. Without loss of generality, we assume that this table is sorted on the week and day attributes. Hence, the query “Which stocks have had continuously rising prices during an entire week?” can be simply evaluated by a scan of the *STOCKINFO* table followed by the StreamJoin operator. The corresponding signature is expressed as follows:

StreamJoin (week, day, stockkey.new = stockkey.old; price.new > price.old).

Hence, the *GroupId* parameter is set to *week*, *StreamId* is set to *day*, and the join is performed on the *stockkey* and *price* attributes. In this way only those tuples of a new stream, i.e. a new day, qualify, that satisfy the condition that for the same *stockkey* (*stockkey.new = stockkey.old*) the price is rising (*price.new > price.old*). Subsequent projection on attributes *stockkey* and *week* finally yields the stocks that have had continuously increasing prices for all successive days of a week.

Assume that the user is further interested in stocks that are chasing one another. In a simplified model, this can be defined by the fact that every time the price of the first stock is going up during an entire week, the price of the second stock is also rising during the subsequent week. Thus, the query can be answered by a self-join followed by the StreamJoin operator as shown by the QEP depicted in Fig. 4. The self-join is on condition that the attribute value *week2* is subsequent to *week1*, given by the expression *week2 = week1 + 1*. The intermediate result *(stockkey1, week1, stockkey2)* expresses pairs of stocks that chase one another in two subsequent weeks. However, the condition for chasing stock prices is that *every time* the first stock is rising, the second is also rising in the subsequent week. This condition is evaluated by the StreamJoin operator. The input is delivered by the intermediate result described above, by setting the parameters *GroupId* to *stockkey1*, *StreamId* to *week1* and by defining the join predicate on *stockkey2*.

In Fig. 4, we have depicted the group defined by the stock *IBM*. This group is constituted of two streams, corresponding to the weeks with rising prices for this stock.

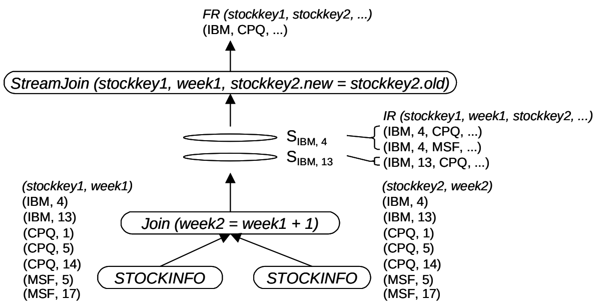

Figure 4: QEP for the usage of the StreamJoin operator in financial time series.

By joining the streams on *stockkey2*, we obtain the stocks that chase *IBM* in *all* subsequent weeks. In this example, the price of *CPQ* is rising every time when the price of *IBM* is rising, while *MSF* is chasing *IBM* only in the fourth week. Thus, *MSF* is eliminated by StreamJoin for the final result.

5. Pattern Discovery in Genomic Databases

Genomic databases assist molecular biologists in understanding the biochemical function, chemical structure, and evolutionary history of organisms. Popular systems for searching genomic databases perform a type of pattern matching over data sets called *sequences*. Efficiency in such exhaustive systems is crucial, since some servers process over 40,000 queries per day [2], and several queries require comparison to over one gigabyte of genomic sequence data. A genomic database contains sequence records that are continuous strings drawn from a specific alphabet, varying from a few characters to several hundred thousand characters in length. During each query task, a new string called *pattern* has to be matched against the old strings. Thereby, the strategy must be able to find statistically significant similarities in the presence of not only varying sequence lengths, but also repetitive subsequences. Contrary to most related work [23] [22], our approach to discovering patterns in a database of genetic sequences is realized within the database engine.

We consider the patterns as being regular expressions of the form $*X_1*X_2*...$, where X_1, X_2 are segments of a sequence made up of consecutive letters, and * represents a variable length of intermediate letters. We treat this variable length as a parameter, called *int_length*. The user is interested in the locations (positions) where a pattern is contained in a given sequence. Suppose that the information corresponding to sequences is stored in a table *SEQUENCE (pos, letter)* and the pattern is stored in a table *PATTERN (letter)*. Please note that this is only a simplified representation. Thus, if e.g. there are several patterns to be analyzed, the *PATTERN* table has to be extended by a *patternkey* attribute. Similarly, in such a general case, the *SEQUENCE* table contains a *sequencekey* attribute as

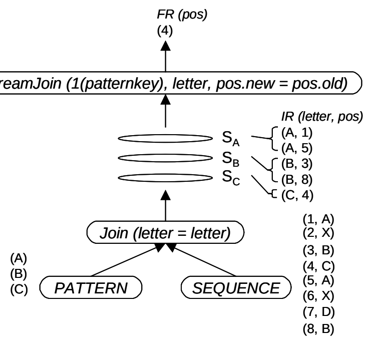

Figure 5: QEP for the usage of the StreamJoin operator in pattern recognition.

well. In the example that follows we assume that the pattern has the form *A*B*C*, with *int_length* being 1, i.e. one intermediate letter is allowed for matching subsequences.

This query can be evaluated in an integrated fashion by using the StreamJoin operator as shown in Fig. 5. Here, each element of the pattern defines a stream, containing all positions of the sequence where this element occurs. We are interested in the portions of the sequence that contain all elements of the pattern in the *given order with the imposed position requirements*. Hence, for a variable length of intermediate letters the join condition for the StreamJoin operator is on the *pos* attribute, expressed by the following parameter:

pos.new <= pos.old + int_length + 1.

In the example from Fig. 5, by substituting *int_length* with 1, we obtain:

pos.new <= pos.old + 2.

The *GroupId* parameter of the StreamJoin operator is set to 1 (or any constant value), since a group is defined by a pattern and in this simple example there is only one pattern involved. In the generalized case the *GroupId* is set to the *patternkey* attribute and *StreamId* is set to the *letter* attribute. In our example we have three streams corresponding to the letters *A*, *B*, and *C* of the pattern. As mentioned, only the sequences satisfying the imposed position requirements qualify. For instance, the tuples *(A, 5)* and *(B, 8)* of the intermediate result *IR* do not survive, since the distance between the letters is greater than 1. The result contains the end positions of the matching sequences found. In this example, the pattern is included in the *SEQUENCE* table from position 1 to 4, hence the result contains the position 4.

6. Performance Evaluation

In the following, we present our preliminary measurement results and first performance assessments for the universal quantification problem.

6.1. Measurements

For the performance evaluation, we have integrated the StreamJoin operator into the MIDAS system. MIDAS [4] is a prototype of a parallel object-relational database system running on a hybrid architecture comprising several SMP nodes combined in a shared-disk manner. We used a 100 MB TPC-D database [21], running on a cluster of 4 SUN-ULTRA1 workstations with 143 MHz Ultra SPARC processors, connected via a Fast Ethernet network. For this database, we have evaluated different possibilities of evaluating the query presented in Section 3.3, namely “Find the suppliers who supply a part that is being sold in all nations of a region.”

We first compared the memory requirements of the StreamJoin and hash-division approaches, on condition that the latter is extended to handle also complex divisors, e.g. by keeping several bitmaps for each quotient candidate in memory. In our database the number of distinct <*partkey*, *suppkey*> combinations is 79.947. Each such combination has to keep a separate bitmap corresponding to each region. In the TPC-D database, the number of regions is 5, each region being constituted of 5 nations. Thus, the memory requirement for the quotient table of the hash-division algorithm is as follows: 79,947 (tuples) $\times$ 16 bits (*partkey*, *suppkey* attributes) $\times$ 5 (bitmaps corresponding to each *region*) $\times$ 5 bits (*nations* per *region*) = 3.2 MB.

The quotient table is permanently needed by the algorithm, thus, if it does not fit into memory, both divisor and quotient tables have to be partitioned, resulting into considerable disk I/O costs. Please recall that the intermediate result size of the StreamJoin approach decreases in each step. During the measurements, the memory consumption of this algorithm averaged to ca. 280 KB. The peak of the memory consumption has been 700 KB. However, this has been measured only for a small time period, corresponding to a single stream. Furthermore, the hash-division algorithm has to consume all of its input before it produces the first output tuple. An additional advantage of the StreamJoin algorithm is that it forwards the result tuples region by region, thus making this approach also more attractive for pipelining as well as interactive use e.g. by OLAP users.

As presented in Section 3.3, the anti-semijoin approach is not suitable for this scenario. Other possibilities [6] for evaluating the all-quantificator are based on counting or set difference. Hence, we compared the StreamJoin evaluation with these two strategies as well.

In this performance evaluation, we have used different QEP variants for the StreamJoin approach. These variants are identical in the way they use the StreamJoin operator. However, they differ in the strategies employed to accomplish the necessary grouping on the *GroupId* and *StreamId* attributes for the input of StreamJoin. This grouping can of course always be accomplished by a cor-

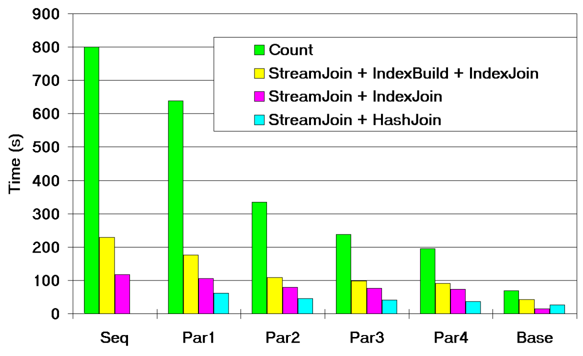

Figure 6: Performance Evaluation.

responding *sort* operator. However, this is not always necessary if the optimizer chooses adequate physical implementations for the operators delivering the StreamJoin input.

In the TPC-D database used, the central table is called *LINEITEM*. The example of Fig. 3 that is already discussed in Section 3.3 is adopted here; the input of the StreamJoin operator is delivered by an equi-join between *LINEITEM* and the computed table *NATION* that delivers the *regionkey* and *nationkey* attributes. Assume that *NATION* is sorted on *regionkey* and *nationkey*. Consider the following evaluation alternatives w.r.t. the join between *LINEITEM* and *NATION*:

- an index-nested-loops join, using an index of the *LINEITEM* table; this is mostly possible, since in data warehouse schemas there are typically indexes on the dimension attributes of the fact table.
- a hash join, the *NATION* table being used as the probing table.

In these cases, the join result is constituted as follows: for each tuple of the *NATION* table a set of tuples *(regionkey, nationkey, ...)* is generated, yielding exactly one stream and containing the parts that have been sold in a given nation of a region. Hence, the necessary grouping of StreamJoin input is already satisfied and no additional sort operations are necessary. In addition to the two possibilities presented above, for our performance evaluation we have also considered a QEP variant where the index for the index-nested-loops join has been built on the fly.

The results of our performance evaluation for the approach based on counting as well as for the StreamJoin variants can be found in Fig. 6. The approach based on set difference took more than 10 hours for only one region, hence it is obviously not competitive. In Fig. 6, we present our measurement results for both the sequential case (called *Seq* in the figure) as well as for the parallel scenarios, called *Par1* (in this case only pipelining has been used) to *Par4*, according to the degree of parallelism used. Hereby, we have only modified the degree of parallelism of the subplans that have been in charge of performing the universal quantification. The rest of the QEPs (called *Base* in Fig. 6) has been left unchanged. In order to be able to asses the speedups correctly, in Fig. 6 we have given the constant costs of these unchanged QEP parts as well.

The performance evaluation shows that the StreamJoin approach outperforms the variant based on counting by factors, even if the index of the *LINEITEM* table is built on the fly. The best results have been achieved for the hash-join variant, as this approach allows a uniform evaluation with minimal I/O costs after the hash table has been built. However, this variant can only be used if the database cache is large enough to hold the tables of both the hash-join and StreamJoin operators. One can see that the only situation where this requirement could not be accomplished was the sequential case, where the entire QEP is evaluated on the same processing node.
As shown in Fig. 6, after subtracting the constant base costs, quasi-linear speedups have been achieved. This demonstrates the good parallelization potential of the StreamJoin approach, thus further increasing performance.

6.2. Performance Assessment

Please note that the preliminary performance evaluation presented in the previous section refers only to the all-quantification facility of the StreamJoin operator. Our aim was to show that this basic functionality already presents good results and thus is worthwhile to be considered as a primitive in the database engine. Important characteristics of this operator are the low memory consumption and avoidance of intermediate result materializations, all resulting in reduced I/O.

However, as presented in the previous sections of this paper, as well as in [17] [15], StreamJoin is able to express much more complicated application-specific functionality as well. It can be efficiently integrated into the database engine, as a stand-alone operator or a user-defined table operator [13] [17]. The resulting plans guarantee a non-blocking and uniform data flow among the operators involved. Additionally, StreamJoin can be efficiently expressed and referenced by SQL-like languages [17].

Hence, a parallel execution [16], reliability, and portability are supported.

7. Conclusion

In this paper we have pointed out the importance of stream-oriented processing as a generic execution strategy that is applied by different applications like time series, DNA analysis, data mining, and universal quantification. We have presented a new database operator, called StreamJoin, that provides a resource-effective and efficient strategy to solve the problem of stream analysis within the database engine. Thus, no data transfer between the database and the application is necessary. Taking into account the volume of the involved applications, this aspect alone already results in substantial performance improvements.

The StreamJoin operator interfaces in a natural way the other query processing capabilities of the database engine, thereby being able to make use of all existing facilities and optimizations, such as indexes, sorting etc. Its non-blocking feature can be efficiently used for pipelining purposes. As shown by relevant examples in this paper, adaptability to diverse application domains is provided by means of appropriate parameter settings.
Stream analysis, like most data analysis, is best done in a way that permits interactive exploration. The StreamJoin approach presented in this paper is a novel strategy towards efficiently satisfying such ad hoc queries as well. Thereby, due to its economic memory consumption it is also particularly suitable for multi-user environments.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo. Fast Discovery of Association Rules, in: *Advances in Knowledge Discovery and Data Mining*. Chapter 12, AAAI/MIT Press, 1995.

[2] D. Benson, D. Lipman, J. Ostell. GenBank. *Nucleic Acids Research*, 21(13): 2963-2965, 1993.

[3] S. S. Bhowmick, S. Madria, W. K. Ng. Detecting and Representing Relevant Web Deltas using Web Join. *Proc. ICDCS*, Taipei, China, 2000.

[4] G. Bozas, M. Jaedicke, A. Listl, B. Mitschang, A. Reiser, S. Zimmermann. On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project. *Proc. EUROPAR*, 1996.

[5] F. Cacace, S. Ceri, M. A. W. Houtsma. A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs. *Distributed and Parallel Databases*, 1(4): 337-382, 1993.

[6] J. Claussen, A. Kemper, G. Moerkotte, K. Peithner. Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. *Proc. VLDB*, Athens, Greece, 1997.

[7] U. Dayal: *Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries*. Proc. VLDB, Brighton, 1987.

[8] G. Graefe, R. L. Cole. Fast Algorithms for Universal Quantification in Large Databases. *TODS*, 20(2): 187-236, 1995.

[9] P. Gulutzan, T. Pelzer. *SQL Complete, Really*. R&D Books, 1999.

[10] J. Han, W. Gong, Y. Yin. Mining Segment-Wise Periodic Patterns in Time Related Databases. *Proc. KDD*, New York City, NY, 1998.

[11] M. A. W. Houtsma, A. N. Wildschut, J. Flokstra. Implementation and Performance Evaluation of a Parallel Transitive Closure Algorithm on PRISMA/DB. *Proc. VLDB*, Dublin, 1993.

[12] P. Hsu, D. Parker. Improving SQL with Generalized Quantifiers. *Proc. Data Engineering*, Taipeh, Taiwan, 1995.

[13] M. Jaedicke, B. Mitschang. User-Defined Table Operators: Enhancing Extensibility for ORDBMS. *Proc. VLDB*, Edinburgh, 1999.

[14] L. Molesky, M. Caruso. Managing Financial Time Series Data: Object-Relational and Object Database Systems. *Tutorial VLDB Conf.*, New York City, NY, 1998.

[15] C. Nippl, M. Jaedicke, B. Mitschang. Accelerating Profiling Services by Parallel Database Technology. *Proc. PDPTA*, Las Vegas, 1997.

[16] C. Nippl, B. Mitschang. TOPAZ: A Cost-Based, Rule-Driven, Multi-Phase Parallelizer. *Proc. VLDB*, New York City, 1998.

[17] C. Nippl, A. Reiser, B. Mitschang. Extending Database Functionality to Support Frequent Itemset Processing. *Technical Report*, Technical University of Munich, 2000.

[18] C. Nippl, A. Reiser, B. Mitschang. Conquering the Search Space for the Calculation of the Maximal Frequent Set. *Technical Report*, Technical University of Munich, 2000.

[19] S. Rao, A. Badia, D. v. Gucht. Providing Better Support for a Class of Decision Support Queries. *Proc. SIGMOD*, Montreal, 1996.

[20] D. Schneider. The Ins and Outs of Data Warehousing. *Tutorial on the VLDB Conf.*, Athens, 1997.

[21] Transaction Processing Council. *TPC Benchmark D*. Standard Specification, Revision 1.3, 1997.

[22] T. Tsunoda, M. Fukagawa, T. Takagi. Time and Memory Efficient Algorithm for Extracting Palindromic and Repetitive Subsequences in Nucleic Acid Sequences. *Pacific Symposium on Biocomputing*, 1999.

[23] H. Williams, J. Zobel. Indexing Nucleotide Databases for Fast Query Evaluation. *Proc. EDBT*, Avignon, 1996.

[24] M. J. Zaki. Efficient Enumeration of Frequent Sequences. *Proc. CIKM*, Bethesda, 1998.