Time-completeness trade-offs in record linkage using Adaptive Query Processing

Roald Lengu [◇], Paolo Missier [*], Alvaro A. A. Fernandes [*], Giovanna Guerrini [◇], Marco Mesiti [♮]

[◇]*DISI, Università di Genova, Italy*
rlengu@gmail.com, guerrini@disi.unige.it

[*]*School of Computer Science, University of Manchester, UK*
pmissier@cs.man.ac.uk, alvaro@cs.man.ac.uk

[♮]*DiCo, Università di Milano, Italy*
mesiti@dico.unimi.it

***Abstract*— Applications that involve data integration among multiple sources often require a preliminary step of data reconciliation in order to ensure that tuples match correctly across the sources. In dynamic settings such as data mashups, however, traditional offline data reconciliation techniques that require prior availability of the data may not be applicable. The alternative, performing similarity joins at query time, is computationally expensive, while ignoring the mismatch problem altogether leads to an incomplete integration. In this paper we make the assumption that, in some dynamic integration scenarios, users may agree to trade the completeness of a join result in return for a faster computation. We explore the consequences of this assumption by proposing a novel, hybrid join algorithm that involves a combination of exact and approximate join operators, managed using adaptive query processing techniques. The algorithm is optimistic: it can switch between physical join operators multiple times throughout query processing, but it only resorts to approximate join operators when there is statistical evidence that result completeness is compromised. Our experimental results show that sensible savings in join execution time can be achieved in practice, at the expense of a modest reduction in result completeness.**

I. Introduction

The rise in prominence of rich Internet applications provides new opportunities for on-the-fly integration of data from sources that are selected as a result of interactive user exploration. The problem of record linkage [10] is at the heart of these data integration scenarios, where different and autonomously maintained tables are joined on the expectation that the values of some common attributes match, at least approximately. When two customer databases that belong to different organisations are merged, for example, it is reasonable to expect that the common customers can be found by means of a join. Unless those customers are identified in exactly the same way in both tables and in all instances, however, the result will in general be incomplete.

The term "record linkage" denotes a class of algorithms that attempt to identify pairs of records that are meant to represent the same real-world entity, despite minor differences in the values of their attributes. In this paper we refer to these records as *variants* of each other (others, e.g., [5], have called them fuzzy duplicates). Common to the broad variety of existing approaches to record linkage (see for instance the surveys in [2], [3]) is the definition of a similarity function $sim(r_1, r_2)$ that applies to pairs of records r_1 and r_2, along with decision rules that classify each pair $\langle r_1, r_2 \rangle$ as either a match or a non-match based on the similarity value, e.g. "if $sim(r_1, r_2) > \theta$ then match", where θ is a pre-defined threshold.

Performing record linkage on the tables ensures that the results of subsequent join queries are as complete as possible. This, however, assumes that the tables are available prior to their deployment as part of an application. Table analysis is required for at least two reasons: firstly to tune the decision rules (i.e., to find a suitable value for θ), and secondly to reduce the computational complexity. Tuning is required to improve the performance of the linkage process, i.e., to reduce the number of false positives (i.e., the number of records pairs erroneously reported) and false negatives (the number of true record pairs that the algorithm fails to match). This in general requires the ability to scan the tables. Regarding complexity, note that the need to measure the similarity of each pair of records in two tables of size n involves n^2 comparisons; this complexity can be reduced using *blocking* techniques, whereby records are first partitioned into coarse-grain clusters, so that pairwise comparison is only performed separately within each cluster. Again, this requires that the tables be pre-processed prior to linkage.

Advance access to the tables, however, is increasingly becoming an unrealistic assumption, for instance in *mashup*-style integration scenarios, where two or more data sources are integrated on-the-fly, often by a third party who has no control over either source, or when the data to be joined is a continuous stream. Here linkage is indeed required, because it seems unreasonable to expect a perfect match among values in the two sources; but it can only be performed at query time, using a similarity, or *approximate* join algorithm such as SSH join [4]. Here the similarity function and the match decision rules are embedded into the physical join operator. In this case, however, the inevitable $O(n^2)$ complexity results in high query response times.

The research presented in this paper stems from the observation that, in some scenarios, users may be willing to accept

a less complete result in return for faster join computation. Consider for example a mashup-based integration, where an organization collects data from various insurance companies into a large table of car accidents that have occurred nationwide over a period of time, and that is updated frequently. This data is then overlaid onto a map based on the accidents location, in order to visualize "accidents hot spots". Suppose that the geographical information is obtained by joining this table –itself collated from various sources, with a reference table containing an atlas of all streets in every city, along with their precise map location. Here performing a similarity join seems appropriate, since there is no guarantee that street names in the accidents table match exactly those in the reference atlas. On the other hand, a full-fledged similarity join would prove expensive to compute, and furthermore, such computational cost may not be justified, as the proportion of mismatching accident locations, which is not known a priori, may turn out to be modest. In this situation our organization may be willing to trade the completeness of the result, i.e., the number of accidents laid on the map, in return for a faster visual presentation (the experimental evaluation presented in Section IV makes use of data from this example).

A. Goals and Contributions

In this paper we explore such trade-offs. Specifically, we propose a hybrid join algorithm that employs a combination of exact and approximate join operators and that is able to dynamically switch between the two, depending on a time series of estimates of the number of variant records that occur in the tables. As we show in Section III-B, such estimates are indeed available whenever a parent-child relationship between the two tables is expected, but it is not enforced through a constraint, and indeed the presence of variants reduce the number of actual matches. In the example mentioned earlier, for instance, we expect each accident record to match one record in the reference street atlas when there are no variants. In this case the presence of variants can be detected based on the divergence between the observed and the expected join result size, at various points during the computation.

Current results from the adaptive query processing (AQP) framework [8] show how, in certain cases, the query processor can use these estimates to modify the query plan during execution, namely by replacing a physical operator with another that performs the same function [12], [14], [15], [18], [19], [28], [25]. This idea has proven viable for pipelined query plans [11], primarily as a dynamic optimization technique to improve the performance of a complex query, in cases where the initial query plan produced by the optimizer proves inefficient.

In this paper we explore the applicability of the AQP framework to our problem.Our goal is to control the trade-off between the overall cost of the join vs. the completeness of the result, i.e., by switching to an (inexpensive) exact join when no variants are detected, and returning to an (expensive) approximate join when they are. To the best of our knowledge, this is the first attempt to apply a well-known query optimization technique to achieve a balance between the cost of the query, and the completeness of the result. The adaptive strategy seems well-suited to achieve this goal by means of a statististical model of the expected join result size at any point during the computation; this is critical to being able to detect unexpected variants in the tables, and therefore to react by switching join operators. Our experimental results show that the technique is indeed viable, and that sensible savings in join execution time can be achieved at the expense of a modest reduction in result completeness.

The paper offers the following specific contributions:

- an adaptive join processing algorithm based on exact and approximate symmetric hash joins. In particular, we have modified the SHJoin algorithm [4] to ensure that operators can be switched safely at certain well-defined points during the computation (Section II);
- an instantiation of the generic Monitor-Assess-Respond framework for adaptive query processing that uses the symmetric join operators, and its formalization (Section III);
- a definition of criteria to measure the algorithm's performance, along with experimental results (Section IV).

A discussion of related work (Section V) concludes the paper.

II. Exact and Approximate Join Operators

We begin with an introduction to our model for adaptive join processing, inspired by the generic *monitor-assess-respond* (MAR) framework for functional decomposition of autonomic systems [21]. As mentioned, we adapt a model previously designed to address query optimization problems [14], [15], to address issues of record linkage.

We model the query processor as a state machine, where each state represents one configuration of the query plan that is currently being executed. In our case, all the processor can do is to dynamically switch between two join operators, one exact and one approximate. Therefore, in the simplest model the processor has only two states, corresponding to each of the operators, and each operator switch corresponds to a state transition (in Section III-D we will generalize the processor model to multiple states).

The control loop shown in Fig. 1 defines the overall adaptive model that we are going to use. The *monitor* periodically obtains the values for some observable quantities from the query processor; the *assessor* performs an analysis of those values, in order to determine whether a change of operator is required; when this is the case, the *responder* performs a state transition, which is enacted by the processor as an actual operator switch.

In the rest of this section we describe the specific operators used by the processors in our implementations, and the switching mechanism. Details on the monitored variables and on the assessor's decision logic are presented in the next section.

A. Requirements for Join Operators

The choice of physical join operators is dictated by two main requirements. Firstly, we need to make sure that the state

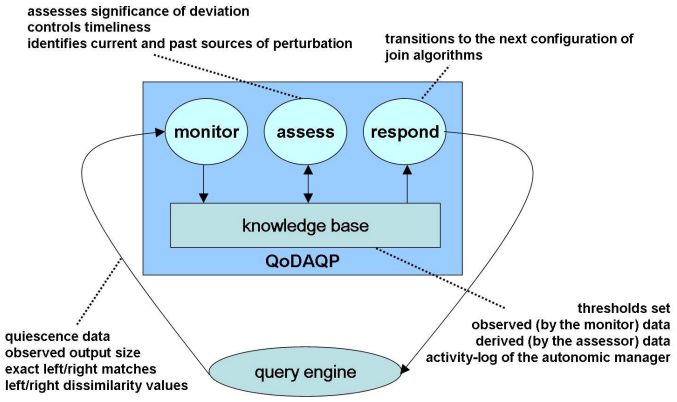

Fig. 1. The Monitor-Assess-Respond adaptive framework

of the join execution can be transferred from one operator to the next without loss of data, i.e., of partial results, and at specific points during the execution.

As formally shown in [11], a sufficient condition for a "safe" switch, i.e., one where we can correctly compute the remainder of the join result after the switch without re-processing the tuples seen so far, is that the operators expose a particular state, called *quiescent*. In iterator-based evaluation [16], an operator implements three operations, i.e., OPEN(), NEXT() and CLOSE() (see Fig. 2). An actual execution trace is a path through the diagram. As shown in [11], some of the states N' in the execution trace, i.e., the state the join algorithm is in when a call to NEXT() has concluded, are indeed quiescent states, making iterator-based algorithms a suitable choice for our purposes. The precise characterization of which N' states are quiescent states is specific to each join algorithm.

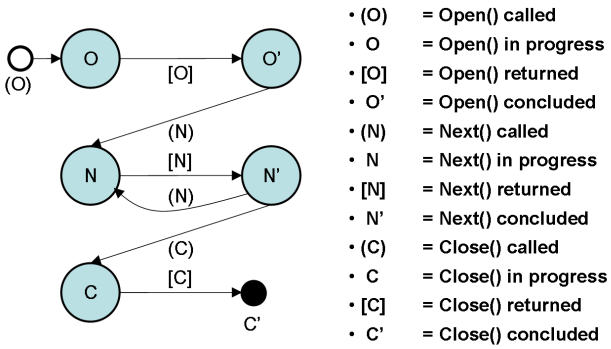

Fig. 2. State-Transition Diagram of an Iterator

A second requirement derives from the need to support joins among tables that are actually data streams. As mentioned in the introduction, the streaming scenario is one where a priori analysis of the tables involved is not feasible, making adaptivity a particularly attractive option. For this reason, we use pipelined operators in our implementation.

Hash join operators can be made to satisfy these requirements. In particular, we have adopted a traditional symmetric hash join [31] for the exact operator, denoted SHJoin, because it is the most used for streams, and we have implemented a pipelined version of the SSJoin algorithm proposed in [4], denoted SSHJoin and briefly presented in next section.

SHJoin is the pipeline version of the traditional hash join. It uses two hash tables that are built in parallel while reading the tuples from both inputs. When reading a tuple from one input, it is probed through the use of the hash table of the other input to identify matches. Therefore, results can be returned at any point of the join execution, using the partial information contained in the tables, without having to wait for the input total exhaustion.

B. The SSHJOIN Algorithm

SSHJoin is our pipelined, symmetric hash re-implementation of the SSJoin operator [4]. For reasons of space, here we only give a short overview; a more detailed description is available [23].

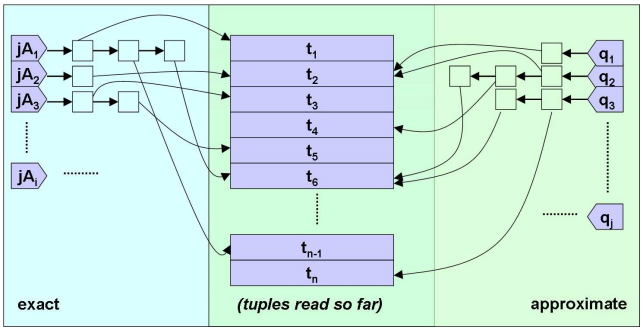

Fig. 3. SHJoin and SSHJoin State, per Operand

Fig. 3 shows the hash data structures maintained by SHJoin and SSHJoin, on the left and right, respectively, for each of the two joining tables (only one is shown for simplicity). Each algorithm operates by scanning each of the tables in turn, one tuple at a time[1]. The main difference between the two is that, while SHJoin computes a hash key of the join attribute values read so far, the entries q_i on the right-side of Fig. 3 are the

[1]In line with other adaptive join techniques that use hash join (e.g., [28], [15]), we assume that join state can be held in main memory.

hashed values of all the q-grams seen by SSJoin. The set of q-grams of a string s, denoted $q(s)$, is the set of all substrings obtained by sliding a window of width q (typically, $q = 3$) over s. In this work we use q-grams to measure string similarity, namely by computing the *Jaccard Coefficient*[2]:

$$sim(s_1, s_2) = \frac{|q(s_1) \cap q(s_2)|}{|q(s_1) \cup q(s_2)|}$$

In this case, the bucket for any q-gram q contains references to each scanned tuple that includes q.

Since the algorithms are symmetric, they maintain two hash tables, one for each of the two joining tables (left and right inputs). Thus, every tuple from each table is both inserted into one of the hash tables, and also used to probe the other. SHJoin supports the traditional iterator-based, pull-on-demand, pipelined NEXT() operation. According to [11], in this case a quiescent state is one in which the operator has concluded a call to NEXT() and the last tuple read from either the left or the right input has been joined with all the tuples in the hash table for the right or the left input, resp., that it matches. In SHJoin, NEXT() is computed as follows:

a) If there is an outstanding *probe* tuple for which not all matches have been returned yet, i.e., the state of the operator is not quiescent, then the next match for that probe tuple is returned.
b) If there are no outstanding tuples, i.e., we are in a quiescent state, the operator reads a new tuple, inserts it into the appropriate hash table, and uses it to probe against the other hash table.

SSHJoin follows the same basic behaviour, but the result of one invocation of NEXT() when probing with tuple t and joining, say, on attribute A, is the set of tuple pairs $\langle t, t' \rangle$ such that $sim(t.A, t'.A) > k$. Here k is a pre-defined threshold. This involves computing all q-grams for $t.A$, and probing the q-grams hash table (right-hand side of Fig. 3) with each of them. The tuples t' that are retrieved at least k times are returned as part of the match.

The main differences between SSHJoin and SHJoin can be summarized as follows:

1) an additional operation, i.e., obtaining the q-grams in the join attribute of the tuple, is needed before any insertion can be made in the corresponding hash table;
2) an insertion of a tuple in a hash table additionally requires the insertion of a pointer to that tuple for each q-gram appearing in the join attribute of the tuple;
3) when probing a hash table on q-grams with t, the operator builds a set $T(t)$ of references to tuples t' that share at least one q-gram with t, and it associates a counter $c(t')$ of the common q-grams with each $t' \in T(t)$;
4) the result consists of pairs $\langle t, t' \rangle$ where $t' \in T(t)$ and $c(t') \geq k$.

[2]Other similarity functions based on q-grams can be exploited, see [5] for example.

We also use the constraint $c(t') \geq k$ to optimize on computing $T(t)$, as follows. Suppose $|q(t)| = g$. SSHJoin considers each q-gram in reverse frequency order. For the first $g - k + 1$ q-grams of t, the tuples containing that q-gram (obtained through the hash table) are inserted into $T(t)$ with a counter set to 1. If the q-gram already occurs, the counter is incremented. For the remaining q-grams of t, the counters of the tuples where they occur, and which belong to $T(t)$, are incremented (but no new tuples are inserted). The frequency of a q-gram is simply the number of tuples in the hash table which contain it; this number is saved along with the q-gram.

C. Cost Implications

We now present an analysis of the computational cost due to adaptivity, namely *(i)* the relative additional cost of executing one or more steps using SSHJoin over that of SHJoin, and *(ii)* the overhead cost of switching operators.

Relative Computational and Space Costs: SSHJoin is costlier than SHJoin both in terms of processing time and memory requirements. Concerning time, the increase in complexity can be estimated by looking at the differences between SHJoin and SSHJoin w.r.t. the cost of operations (1)-(4) described earlier in this section. These costs are summarized in Table I, where: $|jA|$ is the average length in characters of a join attribute value jA; q is the q-gram size (thus, the number of q-grams is $|jA| - q + 1$); B^{ex} is the average length of the bucket for SHJoin; and B^{ap} is the average length of the bucket for SSHJoin (i.e., the number of tuples containing a q-gram that is hashed to that bucket).

B^{ap} can be estimated as $B^{ex} * (|jA| + q - 1)$, while $|T(t)|$ is certainly less than $B^{ex} * (|jA| + q - 1)^2$. If we consider each transition between quiescent states in SSHJoin/SHJoin (corresponding to the computation of all the matches for a tuple), then the ratio between the two costs for each such transition is in $\mathcal{O}((|jA|+q-1)^2)$, i.e., quadratic in the number of q-grams in jA.

Concerning space, let n be the number of tuples processed so far from one operand, let s be the average space required for a tuple, and let p be the space required for a pointer. For both join algorithms, the tuples read so far are maintained only once. Thus the space required is $n * s$. For each operand, the space required by the SHJoin hash table is $n * p$ whereas the space required by SSHJoin hash table is $n * (|jA| + q - 1) * p$.

Cost of Switching Operators: The main overhead cost incurred during a switch is due to the need to update the hash tables. The tuples processed so far for both operands are kept in main memory, together with the hash tables needed by the algorithms. The hash table that is used by the current algorithm is up-to-date, whereas the other lags behind, as it only contains the tuples processed until the previous switch point. The *pessimistic* approach of maintaining up-to-date both hash tables has not been considered because it imposes an overhead on the exact case, which we assume to be the cost-effective option in most circumstances.

	Operation	SHJoin	SSHJoin
(i)	obtain q-grams	–	$\|jA\|$
(ii)	update hash table	1	$\|jA\| + q - 1$
(iii)	compute T(t) and associated counters		$(\|jA\| + q - 1) * B^{ap}$
(iv)	find matches	B^{ex}	$\|T(t)\|$

TABLE I
COST OF SSHJOIN AND SHJOIN OPERATIONS

Therefore, at each switch point, we need to update the appropriate hash table by inserting the tuples that were processed since the last switch. This means that when we switch from exact to approximate, the hash table on q-grams is updated with the tuples that were being processed by the exact algorithm, and similarly, when we switch from approximate to exact, the hash table on attribute values is updated with the tuples that were being processed by the approximate algorithm. Thus, the switch cost only depends on the number of tuples seen *since the last switch*, rather than on all tuples scanned since the start of the join computation.

III. ADAPTIVE JOIN PROCESSING FOR VARIANTS

We now discuss how the two operators just presented are used to implement the actual adaptive behaviour defined by the general MAR control loop of Fig. 1. We present an overview of the algorithm and its formal details.

A. Algorithm Overview

The overall algorithm consists of periodic activations of the control loop, every δ_{adapt} steps of the symmetric join. At any one time, one of the two operators is active. One step consists of the sequence of elementary operations that move the active operator from one quiescent state to the next, as described above. One activation begins with the *monitor* reading the current size of the join result. The *assessor* computes the estimated result size at that point in the join, and determines whether the divergence between observed and expected result sizes is statistically significant. With this information, the *responder* determines the next state for the query processor; since the current operator is by definition in a quiescent state, the state transition may involve an operator switch at this point.

Initially the system assumes, optimistically, that there will be no variants and therefore the exact join operator is used in the initial state. As variants occur in either of the two tables, their effect manifests itself as a reduction in the observed number of matching tuples. As soon as the lag between observed and expected result size represents sufficient statistical evidence to trigger a reaction by the responder, the approximate join is activated. In turn, this has the effect of reducing the lag, because we are now guaranteed that variants will be detected[3]. The monitor now observes a sliding window of similarity values between each tuple pair being matched. A long sequence of consistently high similarities is taken as an indication that variants no longer occur, prompting the algorithm to return to an exact join operator.

The key point to note is the statistical significance of the observed deviation from the expected course of events. This means that, depending on the relative frequency of the actual variants in the table, the deviation may grow at different rates. In particular, when variants are rare and sparse, the control loop will "lag behind" and will respond slowly. This is an expected behaviour and is part of any adaptive strategy, which is necessarily based on estimates, in line with the general adaptivity framework. Furthermore, as noted in the literature [8], the success of the strategy relies on the accurate tuning of the thresholds and parameters involved. Dynamically finding the best setting for these thresholds and parameters is a hard optimization problem which we (in line with other threshold-based AQP proposals (e.g., [28], [18], [15]) do not address directly but only consider by means of an empirical exploration of the space of available settings (in Sec. IV).

We now describe in detail the process.Specifically, we describe the variables observed by the *monitor*, the logic of the *assessor*, and the state machine controlled by the *responder*.

B. Estimation of Result Completeness

As mentioned in the introduction, the monitor component of the adaptive strategy is based on the assumption that a parent-child relationship is expected between the two input tables, a common case exemplified by the car accidents scenario presented earlier. Under this assumption, the expected result size at the end of the join is, of course, the size of the child table, i.e., each tuple in a child table S matches exactly one in the parent table R. Furthermore, suppose that there are no variants anywhere, and that at some step of a symmetric hash join $n < |R|$ tuples have been scanned. The probability that any given tuple in S has already found its match in R is the same as the probability that the corresponding tuple in R has already appeared among the top n tuples, i.e., $p(n) = \frac{n}{|R|}$. By extension, therefore, the observed result size after scanning n tuples, denoted O_n, can be modelled as a sequence of n independent Bernoulli trials, i.e., as a binomial random variable with parameters n and $p(n)$: $O_n \sim bin(n, p(n))$.

Therefore, the problem of detecting a statistically significant discrepancy between the expected and observed result size after n tuples, reduces to the problem of deciding whether an observation $\bar{O}_n$ is an outlier with respect to its distribution. Outliers are defined using a threshold θ_{out}, namely $\bar{O}_n$ is an outlier iff

$$P_{n,p(n)}(\bar{O}_n \leq O) \leq \theta_{out} \quad (1)$$

[3]As we point out later, past variants can be matched in addition to variants that occur further down the table.

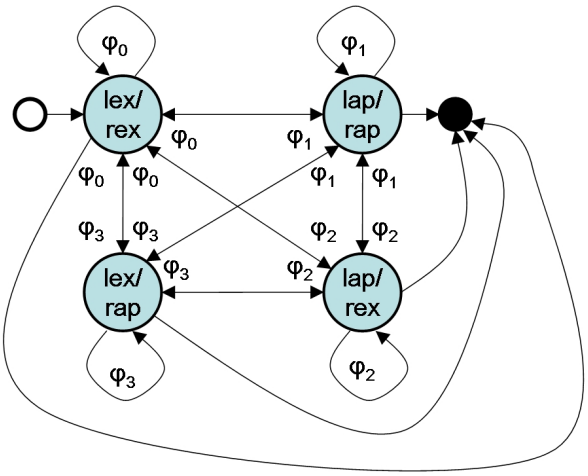

Fig. 4. State machine for adaptive join control

where $P_{n,p(n)}(.)$ is the cumulative distribution function for a binomial with parameters $n, p(n)$.

With reference to the MAR framework, the monitor provides values $\bar{O}_n$ every δ_{adapt} steps, while the assessor computes $P_{n,p(n)}(\bar{O}_n \leq O)$. Note that $P_{n,p(n)}(.)$ changes at every step, i.e., P effectively represents a whole family of functions, and its value at step n cannot be used to compute the value at step $n+1$. In our experiments we have manually tuned parameter δ_{adapt} to achieve a balance between the overhead incurred in computing the cumulative distribution function, and the granularity of the assessment.

C. Identifying the Source of Perturbation

So far we have assumed, implicitly, that the query processor employs either of the two join operators on both inputs. In a symmetric hash join, however, we may also choose to use an exact join when scanning from the left input, while using an approximate join when scanning from the right input (and vice versa). In this hybrid configuration, each tuple read from the left is used to probe a SHJoin hash table on the right, while a tuple read from the right is used to probe the SSHJoin defined on the left input, as explained in Sec. II-B.

This is a useful property. Suppose that, in addition to statistically detecting the presence of variants in the table, we are also able to determine *in which of the two inputs* the variants appear, for example in the left but not in the right. We could then adopt a hybrid configuration where tuples read from the left are matched approximately, while those from the right are matched exactly. Intuitively, this leads to a more accurate use of the two operators.

In order to detect the origin of variants, we add a flag to each scanned tuple in each of the inputs, to denote that the tuple has been successfully matched (at least once). That is, initially the flag is set to false; we set the flag to true if, when probed for an exact match, the tuple matches. Now, assume a tuple t_3 is read off the right input that, through the use of an approximate join, is found to match with a tuple t_2 stored in the left hash table. Now if t_2 has its flag set to true, this means that some t_1 in the right input exists that previously matched t_2 exactly. Therefore t_3, a variant of t_2, is also a variant of t_1 and unless t_1 and t_2 are faulty in identical ways, we can also conclude that it is t_3, rather than t_2 that prevents the exact match between the two. Thus we have been able to conclude that the right input is a source of variants. Of course, if t_2 has not been seen before, and in particular it has not been matched exactly with any tuple from the right input, then we would not be able to glean any evidence from its approximate match with t_3. This is not a problem, however, since in the absence of specific evidence, the algorithm simply assumes the default case, i.e., that variants occur in both tables.

D. State Machine for Adaptive Control

The complete state machine managed by the responder component, that describes the adaptive behaviour of our algorithm, takes account of the hybrid configurations just discussed, and thus it consists of four states, shown in Fig. 4.

Each state represents one of the four possible combinations: (a) in state lex/rex (short for "left exact, right exact") the exact join is used for both the left and the right inputs; (b) in lap/rap ("left approximate, right approximate") the approximate join is used for both the left and the right inputs; (c) in lap/rex the approximate join is used for the left input, and the exact join for the right; and (d) vice versa for lex/rap. As mentioned, the algorithm optimistically begins in the lex/rex state.

The complete set of transitions is defined by predicates $\varphi_i(t)$, $i : 0 \ldots 3$, where t indicates the step of the operator at which the responder is activated (recall that this is one of the quiescent states). Informally, the transitions characterise the following circumstances:

- $\varphi_0(t)$: there is no evidence that tuples from either of the inputs include a statistically significant number of variants. Note that this accounts for two scenarios. Firstly, no variants have ever been detected (this is the transition from lex/rex onto itself); and secondly, the algorithm has at some point reacted to variants, by moving into one of the other states, but recent tuples match with very high similarity, an indication that we can revert to an exact join (transitions back to lex/rex from other states);
- $\varphi_1(t)$: there is evidence that tuples include a statistically significant number of variants, and it is not possible to determine which of the inputs is responsible for the loss of matches;
- $\varphi_2(t)$: there is evidence that tuples include a statistically significant number of variants, and furthermore we can determine that they are located in the left input;
- $\varphi_3(t)$: symmetrically, the variant tuples are located in the right input.

E. Definition of State Transitions

Let us now formalize these transitions in terms of monitored variables, threshold values, and predicates that the responder can evaluate.

Symbol	Interpretation
$\sigma(n) \equiv P_{n,p(n)}(\bar{O}_n \leq O) \leq \theta_{out}$	significant probability of discrepancy
$\mu_i(t) \equiv \frac{A_{t,W}}{W} \leq \theta_{curpert}$	unlikely i is currently perturbed
$\pi_i(t) \equiv \sum_{t'<t} I(\mu_i(t')) \leq \theta_{pastpert}$	unlikely i was ever perturbed

TABLE II
PREDICATES COMPUTED BY THE ASSESSOR

Symbol	Interpretation
W	size of sliding window
θ_{sim}	string similarity threshold
δ_{adapt}	number or steps between successive activation of adaptive control
θ_{out}	outlier detection threshold
$\theta_{curpert}$	acceptable current perturbation threshold
$\theta_{pastpert}$	acceptable past perturbation threshold

TABLE III
THRESHOLDS

Monitored Variables: Transitions $\varphi_1(t)$, $\varphi_2(t)$, and $\varphi_3(t)$, i.e., from an exact to an approximate operator (left, right, or both) rely upon the observed result size at step t, $\bar{O}_t$, as mentioned in Sec. III-B. In addition, transitions $\varphi_0(t)$ from any state other than lex/rex require the ability to recognize that exact operators may be adequate after a portion of the join has been executed using the approximate operator. For this purpose we use a sliding window of size W, applied independently to each input table, and count the number of approximate matches observed within the interval $[t - W, t]$. We denote this number by $A_{t,W}$.

Note that the monitor also reports on the number of steps t executed so far by the join.

Assessor Predicates and Thresholds: The assessor uses the monitored variables to compute three types of predicates. The first, already introduced in Sec. III-B (Eq. 1), indicates whether or not a statistically significant number of variants are present in any of the tables:

$$\sigma(t) \equiv P_{t,p(t)}(\bar{O}_t \leq O) \leq \theta_{out}$$

where θ_{out} is the threshold used to define outliers.

The second type of predicate, $\mu_i(t)$ with $i \in \{\mathsf{left}, \mathsf{right}\}$ is true iff the relative frequency of observed approximate matches within the most recent window of size W is less than a predefined threshold, $\theta_{curpert}$:

$$\mu_i(t) \equiv \frac{A_{t,W}}{W} \leq \theta_{curpert}$$

Finally, the third type of predicate looks at the entire history of evaluations of $\mu_i(t')$ for any $t' < t$, in order to determine how often in the past a high density of approximate matches have been observed:

$$\pi_i(t) \equiv \sum_{t'<t} I(\mu_i(t')) \leq \theta_{pastpert}$$

where $I(\mathsf{true}) = 1$, $I(\mathsf{false}) = 0$.

In addition, the assessor activates the responder only if the interval between the current step of the execution t and the previous is at least δ_{adapt}. Tables II and III summarize the predicates and corresponding thresholds just described.

Responder Predicates: Based on these predicates, we can now formalize the transitions $\varphi_i(t)$, as follows.[4]

[4]Note that all necessary state transitions can be defined using only a subset of all possible conjunctions of those predicates.

$$\varphi_0(t) = \neg\sigma(t) \wedge \mu_{\mathsf{left}}(t) \wedge \mu_{\mathsf{right}}(t)$$

Intuitively, $\varphi_0(t)$ is true when there is no statistical evidence of variants, nor of the left or the right inputs being currently in a perturbation region. In this case, given the available evidence, using exact joins for both left and right tuples (state lex/rex) is both effective (no matches will be lost) and efficient.

$$\varphi_1(t) = \sigma(t) \wedge \neg\mu_{\mathsf{left}}(t) \wedge \neg\mu_{\mathsf{right}}(t)$$

The σ component accounts for evidence of mismatches, and is specifically responsible for the transition out of lex/rex. The other two components indicate that there is no specific evidence to show that the perturbation originates from either source. In this case, therefore, transitioning to lap/rap is more effective (it guarantees not to miss any variant pairs), at the cost of lower efficiency.

$$\varphi_2(t) = \sigma(t) \wedge \neg\mu_{\mathsf{left}}(t) \wedge \mu_{\mathsf{right}}(t) \wedge \pi_{\mathsf{left}}(t)$$

indicates that there is evidence of (1) variants that are affecting result completeness (σ), (2) the left (but not the right) input being currently in a perturbation region, and (3) the left input having been significantly free of past perturbations. In this situation, using an approximate join to match new left tuples is appropriate (on effectiveness grounds), but we may continue using the exact join to match new tuples from the right input. Thus, this calls for a transition to lap/rex.

$$\varphi_3(t) = \sigma(t) \wedge \mu_{\mathsf{left}}(t) \wedge \neg\mu_{\mathsf{right}}(t) \wedge \pi_{\mathsf{right}}(t)$$

In this last case, symmetric with respect to the previous, the right (but not the left) input is currently in a perturbation region, but the right input has been significantly free of past perturbations. Thus, this calls for a transition to state lex/rap.

IV. EXPERIMENTAL EVALUATION

Our adaptive approach is designed to strike a balance between gain, i.e., the increased size of the join result relative to the size that would be obtained using a purely exact join algorithm, and cost, i.e., the increased computational cost that results from the intermittent use of an approximate join. Therefore, the performance metrics presented here are based on the principle of relative gain with respect to a baseline. Specifically, regarding result completeness we use the number of matched pairs returned by the all-exact join as

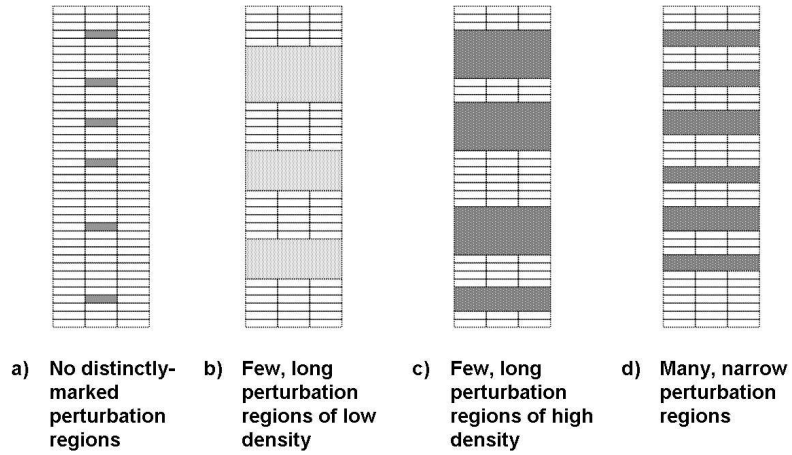

Fig. 5. Perturbation Patterns

a baseline, and count the *additional* number of tuple matched by the hybrid algorithm. Likewise, for computational cost we measure the savings in join computation time achieved by the hybrid algorithm, relative to the execution time in the all-approximate case, used as the baseline.

In this section we report on the experimental evaluation of our proposed technique, expressed using such cost:gain ratio, and measured on test datasets with known perturbation patterns.

A. Generation of Test Datasets

In order to verify the applicability of the developed algorithm in a variety of real situations, we have developed a test data generator that can produce a variety of *patterns of data perturbation*. We remark that considering a range of different possible situations is particular relevant because up to now there is no real benchmark that can be exploited for this purpose (with annotation on the occurrences of variants). The generator can produce a uniform distribution of variants across the length of an input, for example, as well as the presence of perturbation regions (i.e., relatively long regions of variant-rich tuples within the input), interleaved with perturbation-free regions. The latter pattern is designed to simulate various real-life configurations, where batches of data from different sources are collated possibly at different times. Perturbation regions of varying density are created when these sources refer to the same real-world entities using variants. Examples of these patterns are shown in Fig. 5.

From a performance perspective, intuitively we expect better results from datasets in which variants are *not* uniformly distributed, because a burst of variants in one of the two inputs rapidly widens the gap between the observed and the expected result size, allowing for quick detection of any anomaly and hence a timely switch of operators.

In our evaluation we have investigated how different perturbation patterns affect our adaptive strategy. Each pattern is described as a sequence of regions in each of the two joining inputs. Using our generator, we can control *(i)* the *intensity* of the perturbation exhibited by any given region, i.e. the proportion of variants among all tuples in the perturbation region, *(ii)* the *length* of the perturbation region, and *(iii)* the *interval* between two perturbation regions. The results presented here aim to characterize the contexts in which our approach is applicable, i.e., we aim to discover the perturbation patterns that our adaptive techniques contend with well, and the threshold and parameter settings that are required for that performance to be achieved.

Towards this goal, we started by generating a pair of parent-child input tables for each of the four configurations in Fig. 5. Also, we allow variants to appear in both inputs (independently from one another), to account for all possible states in Fig. 4. Throughout the experiments we have set the proportion of variants within an input at a fixed 10%. While our strategy would clearly benefit from higher error rates, this rate is generally accepted as representative of real-world datasets that contain misspellings.

Fig. 5.(a) represents a rather uniform distribution of variants throughout the input, with no distinguishable high-intensity perturbation regions. In this case, we expect a slow accumulation of evidence of statistically-significant discrepancies in the observed result size, and, as a result, a slow reaction to the sparse variants. Fig. 5.(b) captures the situation where low-intensity perturbation regions (light gray) are interleaved with stretches of unperturbed regions. Fig. 5.(c) has a small number of well-distinguished, high-intensity perturbation regions. Finally, Fig. 5.(d) exhibits many high-intensity perturbation regions (having fixed the total variant rate across the entire input, a higher number of variant regions translates to perturbation regions with shorter duration).

Furthermore, we distinguish the case where variants are only present in the child table, from the case where they appear in both tables. Faced with a large number of pattern combinations (each table may be perturbed according to one of the four patterns), we have chosen to focus on the cases where the same pattern applies to both tables. Our results (below) indicate only marginal differences in behaviour across the

patterns, suggesting that we would not have gained additional performance insights by further mixing the patterns. In the following, therefore, we consider eight distinct test cases, namely two (variants in the child, variants in both tables) for each pattern.

Following our introductory example, we have used a parent table containing locations within a country (i.e., all 8082 municipalities in Italy), and a child table containing records of car accidents that occurred in those locations[5]. These are joined on a single string representing location values, e.g., `TAA BZ SANTA CRISTINA VALGARDENA`. A variant value is obtained by introducing a small, one-character variation in the string, e.g., `TAA BZ SANTA CRISTINx VALGARDENA`, resulting in an invalid location. Such edit distance of 1 is enough to guarantee failure of an exact match, but at the same time makes it easy to tune the similarity threshold θ_{sim} in order to control the generation of false positives, i.e., of spurious matches, when using the approximate join. Recall that our goal is not to study the performance of similarity functions, but rather to measure the effectiveness of our adaptive approach under the assumption that the performance of the similarity function on the test data is known in advance.

B. Tuning of Parameters

The effectiveness of the approach is affected both by the setting of the MAR parameters described in Sec. III-D, and by the way in which perturbation regions appear in the inputs. The suite of parameters that are used by the assessor makes for a potentially large space of configurations. The results presented below refer to the best possible configuration for each of the eight test cases described above, obtained by experimentally tuning the setting of these parameters.

Somewhat surprisingly, we have found that the best settings for each parameter oscillate within a small range regardless of the test case. In particular, θ_{sim} was set in such a way that when the join runs exclusively in the lap/rap state, the result size is as close as possible to the expected size, i.e., that of the child table. A value of 0.85 turned out to be appropriate for all test cases.

Similarly, δ_{adapt}, the frequency of assessment, is set empirically by observing the relative gain for different frequencies. A value of δ_{adapt} was deemed adequate. Also, we set $W = 100$. We also found that the algorithm is not very sensitive to the setting of θ_{out}, the threshold used in the σ predicate to trigger a transition from lex/rex. We set $\theta_{out} = 0.05$ throughout. However, variations in $\theta_{curpert}$ and $\theta_{pastpert}$, the thresholds used for predicates π and μ, respectively, result in appreciable variations in the gain/cost ratio. The best settings were found to be $\theta_{curpert} = 2$ and $2 \leq \theta_{pastpert} \leq 5$, depending on the pattern.

[5]These tables were generated by the same generator used in [25] and subsequently in several AQP papers. We thank Volker Markl for kindly allowing us access to it.

C. Measuring Gain and Cost

To assess the relative gain g_{rel}, for each test case, we consider the gap $R - r$ between the result size R obtained by executing the approximate join throughout, and the result size r obtained by executing the exact join throughout. Since our adaptive strategy produces an intermediate result size, $r \leq r_{abs} \leq R$, we express the gain as *the fraction of the gap that has been recovered*:

$$g_{rel} = \frac{(r_{abs} - r)}{R - r}$$

The cost assessment is determined empirically, in agreement with the analysis in Sec. II-C. In particular, the total cost breaks down into (*i*) the cost of performing each step of the symmetric join, when the algorithm is in any one of the four possible states, plus (*ii*) the overhead cost due to all the state transitions. Recall that one step of the algorithm includes all the operations executed between two consecutive quiescent states. We express this total cost as a vector of eight elements, the *state costs* sc_i, plus the *transition costs* tc_i, $i \in \{\text{lex/rex}, \text{lap/rex}, \text{lex/rap}, \text{lap/rap}\}$. In turn, the execution cost sc_i in state i is the product $sc_i = t_i \cdot w_i$ of the number of steps t_i spent in state i, multiplied by the unit cost w_i of a step in that particular state. The weights w_i are determined experimentally, by collecting the actual elapsed times for each step in each possible state. These times are averaged over all experiments; furthermore, since we use the lex/rex baseline case as the best cost, the weights are normalised by the experimental unit cost $w_{\text{lex/rex}}$. These weights are as follows: $[w_{\text{lex/rex}}, w_{\text{lap/rex}}, w_{\text{lex/rap}}, w_{\text{lap/rap}}] = [1, 22.14, 51.8, 70.2]$ This means, for instance, that one step in state lap/rap costs about 70 times as much as one step in state lex/rex. Having set the weights, the actual costs sc_i for a particular test case are determined simply by counting t_i for each state during the execution of that test case.

The transition costs are computed in a similar fashion, as the product $tc_i = tr_i \cdot v_i$ of the number tr_i of transitions into state i throughout the join execution, times the weights v_i, which are determined by observing the actual transition times across all test cases. Once again these times are normalised by considering the unit step cost $w_{\text{lex/rex}}$ as the baseline. The weights v_i are as follows: $[v_{\text{lex/rex}}, v_{\text{lap/rex}}, v_{\text{lex/rap}}, v_{\text{lap/rap}}] = [122.48, 37.96, 84.99, 173.42]$. Thus, e.g., transitioning into state lap/rap has a cost that is equivalent to executing about 173 steps in the baseline state lex/rex.

The total absolute cost of execution is therefore

$$c_{abs} = \sum_i sc_i + \sum_i tc_i$$

Similar to what we have done for the gain, we express this cost in relative terms, by considering the difference between the best possible cost c, *achieved by using the exact join throughout*, and the worst cost C, incurred *when the approximate join is used throughout*:

$$c_{rel} = \frac{c_{abs}}{C - c}$$

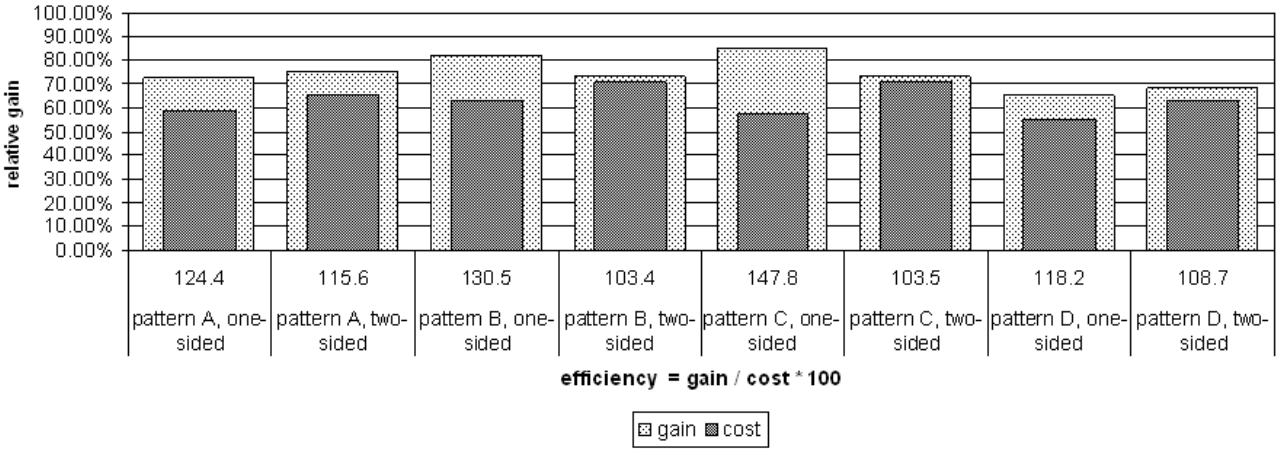

Fig. 6. Gain and Cost across all Test Cases

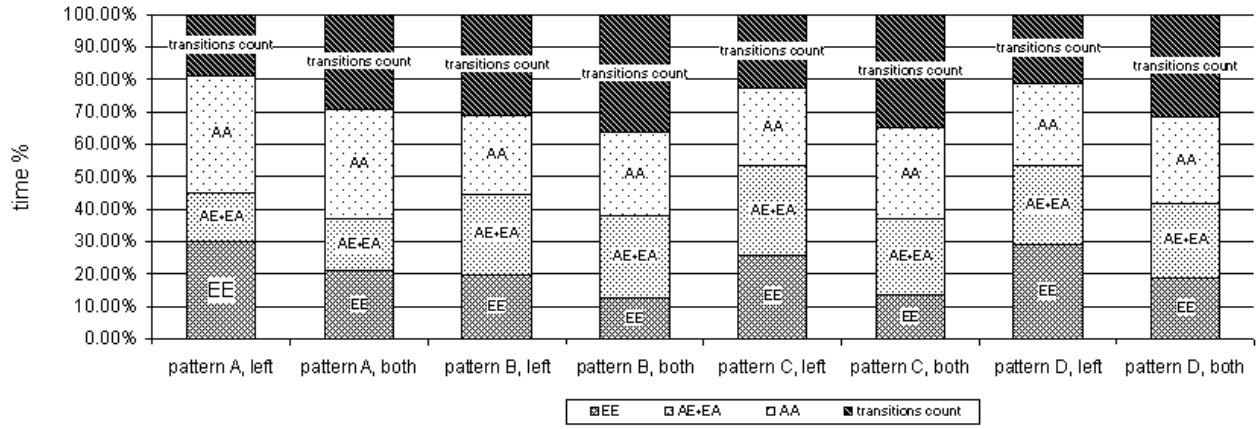

Fig. 7. Breakdown of Relative Execution Times

Note that, in theory, it is possible to have $c_{abs} > C$, i.e., when the costs tc_i prevail, making for a strategy that is less efficient than using the approximate join throughout. In our test cases, however, this is never the case, as described next.

D. Experimental Results and Discussion

Fig. 6 shows the overall gain/cost results across the eight test cases. These are the best results across a range of parameter configurations, as discussed earlier. The efficiency index

$$e = \frac{g_{rel}}{c_{rel}}$$

is reported under each column. As we can see, both relative gains and relative costs are contained within a small interval, regardless of the specific pattern used for the test case, with the higher efficiency being achieved when variants are only present in the child table.

To achieve these results, the algorithm makes use of all four available states to various degrees. Fig. 7 shows a breakdown of the proportion of the time (expressed as the number of steps) spent in each state, as well as of the number of state transitions (for simplicity, we do not break the latter down by specific transition).[6] Notably, the indicated gains are obtained while still spending nearly 30% of the time performing an exact join. Since this fraction, as expected, has a negligible cost compared to the approximate steps, this translates into a substantial reduction in actual costs. This is shown in Fig. 8, where the relative weights above are applied to the raw execution steps reported in Fig. 7. Similarly we note that the transition cost does not contribute significantly to the overall cost.

Note, finally, that the type of perturbation pattern plays no particularly important role in the overall cost, similar to what we observed earlier regarding the overall gain.

We draw two positive conclusions from the analysis above. Firstly, the behaviour of the algorithm does not seem to be significantly affected by the variations in perturbation patterns represented by our test cases. Although we have not explored the broader space of possible pattern combinations, it would be difficult to conclude at this point that the technique works distinctly better, or worse, for some patterns rather than for others. Secondly, the gains accrued using our strategy never incur a cost that is higher than the cost of a purely approximate

[6] In the figure, AA denotes the lap/rap state, EE is lex/rex, AE is lap/rex, and EA is lex/rap.

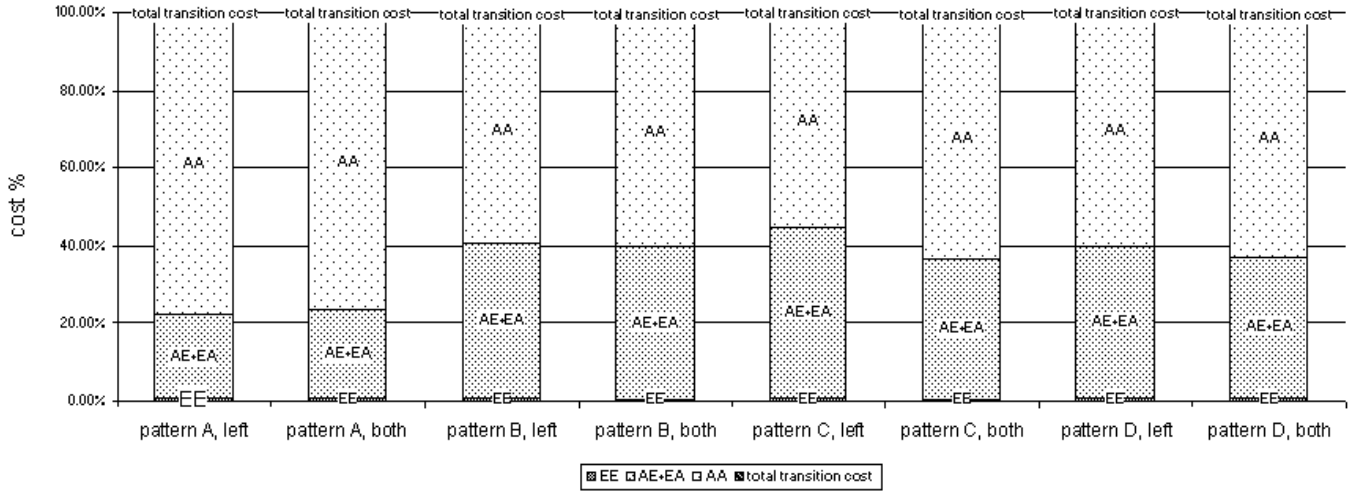

Fig. 8. Breakdown of Relative Execution Costs

join. In other words, the algorithm may choose to transition to the best state at each assessment step without paying an overwhelming price to do so. This suggests the important property that the algorithm may be tuned, possibly under user control, for a target gain in terms of result completeness, while keeping the marginal cost over the exact join baseline within a predictable limit. Additional work will be needed to explore this space of available trade-offs.

V. Related Work

The problem of reconciling database records that represent variants of the same real-world entity, or "record linkage", has a long history, rooted in the practical problem of removing duplicates from large databases prior to conducting data analysis, or to perform integration across data sources. The problem has been studied in detail (see, e.g., [10]) and has spurred the development of a number of research toolkits. These include Potter's wheel [27], Ajax [13], Tailor [9], and BigMatch [32] (see also [2] for a survey of research-oriented tools). Commercial tools are also available, typically from providers of data warehousing and data integration solutions (e.g. Dataflux from SAS, and Vality). These toolkits usually involve data preparation steps, in support of which they provide a variety of utilities, for instance for record normalisation. The need for data preparation, as well as for tuning of the linkage algorithms, is undeniable. In comparison, our approach is at the same time limited, insofar as it focuses exclusively on the matching phase and assumes the existence of suitable similarity functions, and innovative, in that it explores a new direction in data quality research, by recognizing the emerging need to perform on-the-fly, mashup-style integration over data sources that are only made available at the time they are needed. In this respect, we have shown how elements of data quality control can be woven into the fabric of the query processor by exploiting techniques from the AQP area. We are not aware of previous attempts to control data quality adaptively. Note also that our work is much less ambitious than recent proposals for data integration in mashups (e.g., [29]). In only addressing a specific subproblem, however, we do provide a concrete novel approach to address the key requirement (highlighted in [29]) of performing data integration steps dynamically with sufficient accuracy.

We rely on the approximate join technique proposed in [4] but have advanced on that proposal by adapting it to pipelined evaluation. This implies that we use (in the classifications proposed in [26]) a domain-independent, token-based similarity function (as opposed to edit-based, as in [17]). For a survey of approximate join techniques, see [22].

Previous AQP work has focussed on QoS [12], [14], [15], [18], [19], [28], [25] . We build on [11], where the notion of operator replacement in pipelined plans is considered. The notion of asymmetric combinations of joins algorithms (which we build upon) has been discussed in [20] but not, as we do, with a view to achieving an effectiveness:efficiency balance.

VI. Conclusions

In this paper we have addressed the trade-off between results completeness and computational cost, that becomes available when record linkage is performed using a combination of exact and approximate join operators. Such trade-off is interesting in a variety of increasingly common on-the-fly data integration scenarios, e.g. data mashups, where users may be interested in a fast, but incomplete join result and static integration is not an option.

Our hybrid join algorithm builds upon an established framework for adaptive query processing (AQP), whereby the query processor can switch join operators at some well-defined points during the computation, without loss of data. The algorithm involves symmetric hash join operators for exact and similarity-based tuples matching. In particular, we implemented a variation of a known approximate join algorithm, SHJoin, to make it suitable for pipelined processing and thus compatible with the AQP framework.

We have experimentally measured the gain:benefit ratio of our hybrid approach, compared with an all-exact and all-approximate join algorithm, using a suite of synthetically generated datasets that represent a variety of data perturbation patterns. Our results indicate that the algorithm achieves appreciable cost savings, at the expense of modest loss in completeness of the join result.

REFERENCES

[1] P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001.

[2] J. Barateiro and H. Galhardas. A survey of data quality tools. *Datenbank-Spektrum*, 14:15–21, 2005.

[3] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer, 2006.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In Liu et al. [24], page 5.

[5] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *ICDE*, pages 865–876. IEEE Computer Society, 2005.

[6] C.-M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In R. T. Snodgrass and M. Winslett, editors, *SIGMOD Conference*, pages 161–172. ACM Press, 1994.

[7] U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors. *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. IEEE Computer Society, 2003.

[8] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[9] M. Elfeky, A. Elmagarmid, and V. Verykios. Tailor: a record linkage tool box. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, San Jose, CA, Feb. 2002. IEEE Computer Society.

[10] A. K. Elmagarmid, P. G. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, Jan 2007.

[11] K. Eurviriyanukul, A. A. A. Fernandes, and N. W. Paton. A foundation for the replacement of pipelined physical join operators in adaptive query processing. In T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, and J. Wijsen, editors, *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 589–600. Springer, 2006.

[12] S. Ewen, H. Kache, V. Markl, and V. Raman. Progressive query optimization for federated queries. In Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, editors, *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 847–864. Springer, 2006.

[13] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In Apers et al. [1], pages 371–380.

[14] A. Gounaris, N. W. Paton, R. Sakellariou, A. A. A. Fernandes, J. Smith, and P. Watson. Practical adaptation to changing resources in grid query processing. In Liu et al. [24], page 165.

[15] A. Gounaris, J. Smith, N. W. Paton, R. Sakellariou, A. A. A. Fernandes, and P. Watson. Adapting to changing resource performance in grid query processing. In J.-M. Pierson, editor, *DMG*, volume 3836 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2005.

[16] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[17] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In Apers et al. [1], pages 491–500.

[18] W.-S. Han, J. Ng, V. Markl, H. Kache, and M. Kandil. Progressive optimization in a shared-nothing parallel database. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *SIGMOD Conference*, pages 809–820. ACM, 2007.

[19] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In Weikum et al. [30], pages 395–406.

[20] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In Dayal et al. [7], pages 341–352.

[21] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[22] N. Koudas and D. Srivastava. Approximate joins: Concepts and techniques. Tutorial at VLDB, 2005.

[23] R. Lengu, et al. Symmetric Set Hash Join: A Pipelined Approximate Join Algorithm. Technical Report DISI-TR-08-03, DISI, Università di Genova, Via Dodecaneso, Genova, IT, 2008. `ftp://ftp.disi.unige.it/person/GuerriniG/reports/sshjoinTR.pdf`.

[24] L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, editors. *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. IEEE Computer Society, 2006.

[25] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In Weikum et al. [30], pages 659–670.

[26] F. Naumann and K.-U. Sattler. Information quality: Fundamentals, techniques, and use. Tutorial at EDBT, 2006.

[27] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[28] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In Dayal et al. [7], pages 25–36.

[29] A. Thor, D. Aumueller, and E. Rahm. Data Integration Support for Mashups. In *IIWeb*, 2007.

[30] G. Weikum, A. C. König, and S. Deßloch, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 2004.

[31] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77. IEEE Computer Society, 1991.

[32] W. E. Winkler. Overview of record linkage and current research directions. Research Report Statistics #2006-2, Statistical Research Division, U.S. Census Bureau, Washington, DC 20233, 2006. Available at http://www.census.gov/srd/papers/pdf/rrs2006-02.pdf.