

Data Quality support to on-the-fly data integration using Adaptive Query Processing

Paolo Missier², Roald Lengu¹, Alvaro A. A. Fernandes², Giovanna Guerrini¹, and Marco Mesiti³

¹ DISI, Università di Genova, Italy

² School of Computer Science, University of Manchester, United Kingdom

³ DiCo, Università di Milano, Italy

Abstract. In dynamic, on-the-fly relational data integration settings, such as data mashups, there is a need to reconcile values heterogeneity across sources, in order to ensure consistency and completeness of the integrated data. In this scenario, the use of exact joins to match records across sources may lead to incomplete integration, while similarity joins, often advocated as a solution to this problem, is computationally expensive.

In this paper we explore the use of adaptive query processing (AQP) techniques in order to combine exact (fast) and approximate (accurate) joins when performing dynamic integration. The adaptive algorithm uses an an priori expectation of the join result size combined with the monitoring of join progress to statistically determine, at various points during query execution, which join operator should be used. Depending on its configuration, the algorithm can achieve various trade-offs between completeness of the join result, and query execution time.

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.

1 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 [4] 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.

Existing techniques for offline record linkage typically require advance access to, and analysis of the tables. This requirement is increasingly becoming unrealistic, however, 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. Instead, the use of some similarity function as part of the join operation is common in this scenario, to account for small mismatches among attribute values that otherwise refer to the same entity. The result is

a similarity join, i.e., a type of θ -join involving the use of similarity function $sim(s_1, s_2)$ between two strings. Chauduri *et al.* [1] have proposed an extension of symmetric hash join [11] to perform similarity joins efficiently, assuming that $sim()$ is based on the use of q -grams, as explained in more detail in the next section. However the main source of complexity, namely the need to compute the relative overlap of two q-grams, remains.

This extended abstract summarizes recent research that has been more comprehensively presented in [8]. The research stems from the observation that the similarity joins approach is *pessimistic*, in the sense that it assumes that it is always worth paying the full computational cost of using a complex join operator, in return for the guarantee that all potential mismatches among tuples with sufficiently high similarity are caught. We note, however, that in our scenario we have no prior knowledge of how many of tuples in the source tables will fail to match when using an exact join. It may be the case, therefore, that for at least a portion of the tables a fast exact join could be used instead, without loss of joined tuples.

Starting from this observation, in this paper we propose an *optimistic* approach, whereby we assume, initially, that no mismatches will occur at all, i.e., we start off by using a regular exact join. At the same time, however, we put in place a mechanism for detecting mismatches when they do occur, and for reacting to them by switching to a different query execution plan, where the exact join is replaced by a similarity join. We show that this can be accomplished within the well-known adaptive query processing (AQP) framework [3]. AQP techniques have traditionally been applied to the problem of re-optimizing a query plan while it is being executed, by replacing a physical operator in the plan with another that performs the same function [6]. In this work, for the first time (to the best of our knowledge) we apply these techniques in order to offer an alternative to similarity joins when the extent of the mismatches is unknown.

We also show that the added flexibility afforded by our hybrid (exact/approximate) join algorithm results in previously unavailable trade-offs between the completeness of a join result, and its execution efficiency: users can choose a faster execution, at the price of missing more matches, resulting in a lower result completeness; or a more complete join result, at the cost of lower performance.

Such trade-offs may be useful in practical applications. 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. In this case, the accuracy of the overlay map can be sacrificed in part, in return for a faster visual presentation.

In the rest of the paper we describe an adaptive join processing algorithm based on exact and approximate symmetric hash joins. Our experimental results, shown in Section 4, suggest 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.

2 Exact and Approximate Join Operators

Our model for adaptive query processing is based upon the generic *monitor-assess-respond* (MAR) framework for functional decomposition of autonomic systems [7]. In this framework, a *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; and, when this is the case, the *responder* performs a state transition, which is enacted by the processor as an actual operator switch.

While the MAR model was originally designed to address query optimization problems, it also provides a good foundation to model the decisions involved in adaptive record linkage. Specifically, we model the query processor as a state machine and use an instance of MAR to control the state transitions. We deploy the monitor to observe the size of the join result at various stages during the computation of the join; the assessor to encode decision rules for switching between exact and approximate join; and the responder to enact the state transitions in the query processor.

The choice of physical join operators is dictated by two main requirements. Firstly, we need to make sure that the state 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. And secondly, we want to provide support for joins on data streams, as the streaming scenario is one where a priori analysis of the tables involved is not feasible, making adaptivity a particularly attractive option.

To satisfy the first requirement, we rely on the notion of *quiescent* state, i.e., a state of the physical join operator with the property that computation can be interrupted and later resume using a different join operator, without loss of data in the results accumulated so far. Recent results on adaptive query processing [5] show that iterator-based operators do indeed expose quiescent states at well-defined points during the computation; the precise characterization of these states is specific to each join operator. Furthermore, we can show that an implementation of these operators that supports pipelined execution satisfies our second requirement, as well.

We can use variations of existing join operators to satisfy these requirements. Specifically, we have implemented pipelined versions of both a traditional symmetric hash join [11] for the exact join, denoted SHJoin, and of the SSJoin algorithm proposed in [1], denoted SSHJoin, for the approximate join. These operators share the same basic behaviour: they incrementally build two hash tables, one for each input source. At each step one tuple is read from one source, inserted into the corresponding table, and used to probe the other table. The reads alternate between the two sources. As shown in [5], in this case a quiescent state is reached whenever a new tuple from one of the inputs has been added to its table, *and* joined with all tuples in the probe table. The pipelined implementation ensures that partial join results can be consumed as soon as they are produced, without waiting to exhaust the entire input tables.

The difference between the exact and approximate operators is in the way the match is computed. Namely, SSHJoin implements a record linkage algorithm, i.e., a decision procedure that (i) measures the similarity of each pair of records in two datasets, and (ii) declares the pair to be a match if the similarity is greater than a pre-defined threshold, and a non-match otherwise. Different results are obtained by using different similarity

functions with different thresholds. In our case, let $q(s)$, called the q -grams of a string s , be the set of all substrings obtained by sliding a window of width q (typically, $q = 3$) over s . The *Jaccard Coefficient* defines the similarity between two strings s_1 and s_2 as the extent of the overlap of their respective q -grams:⁴

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

The SSHJoin algorithm uses this function and is based on a relational implementation of pairwise comparison of string-based records, whereby $q(s)$ is stored in a relational table, for each s in the database. This reduces the problem of computing the Jaccard coefficient to that of computing the overlap between two sets. As shown in [1], this can be expressed using relational algebra, and therefore implemented using SQL [9]⁵

The similarity between the two operators, along with the availability of quiescent states, makes it possible to transfer partial join results from one to the other, without losing any work. Fig. 1 illustrates the interplay between SHJoin and SSHJoin (only one of the two tables is shown in the figure).

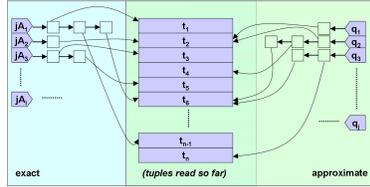


Fig. 1. SHJoin and SSHJoin State, per Operand

join behaviour, the result is a set of tuple pairs $\langle t, t' \rangle$ such that $\text{sim}(t.A, t'.A) > k$. Here k is a pre-defined threshold.

On the left hand side, SHJoin maintains a traditional hash table, indexed on attribute A , with pointers to records in the data table, while the hash table maintained by SSHJoin (on the right in the figure) maps each q -gram to the set of tuples in which it appears. The table is updated when a new tuple is read, and its q -grams are computed. When this table is probed, as part of the standard hash

3 A hybrid adaptive Join Processing algorithm

We now discuss how the two operators just presented are used to implement the actual adaptive behaviour defined by the general MAR control loop. The overall algorithm consists of periodic activations of the control loop. 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

⁴ Other similarity functions based on q -grams can be exploited, see [2] for example.

⁵ Other approaches, not considered here, have been proposed for computing the Jaccard coefficient. *Set hashing*, for example, has been used to succinctly represent the set of tuples containing a given substring[?].

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. 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.

We now describe the variables observed by the *monitor*, the logic of the *assessor*, and the state machine controlled by the *responder*.

3.1 Estimation of Result Completeness

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. With 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 $p(n)$: $O_n \sim \text{bin}(n, p(n))$.

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

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

where $P_{n,p(n)}(\cdot)$ 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 $P_{n,p(n)}(\bar{O}_n \leq O)$ ⁶.

3.2 State Machine for Adaptive Control

We observe that, by the symmetry of each of the two join operators, we can afford to choose to use q-grams-based comparison on each of the two hash tables, independently of the other. This means that, in addition to employing either the exact or the approximate join on both tables, we can also choose to use an exact comparison on one table, which ignores q-grams, while at the same time using an approximate, q-gram-based comparison on the other table.

Thus, the complete state machine managed by the responder component, that describes the adaptive behaviour of our algorithm, includes four, rather than two, possible

⁶ In our Java-based implementation, the cdf is computed using the Apache math-commons package.

hybrid configurations, as, shown in Fig. 2. 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**. The algorithm optimistically begins in the **lex/rex** state.

The complete set of transitions is defined by predicates $\varphi_i(t)$, $i : 0 \dots 3$, where t indicates the step of the operator at which the responder is activated. Informally, the transition condition $\varphi_0(t)$ indicates that there is no evidence that tuples from either of the inputs include a statistically significant number of variants, while $\varphi_2(t)$ and $\varphi_3(t)$ are triggered when there is evidence that tuples include a statistically significant number of variants, and furthermore we can determine that they are located in the left (resp. right) input.

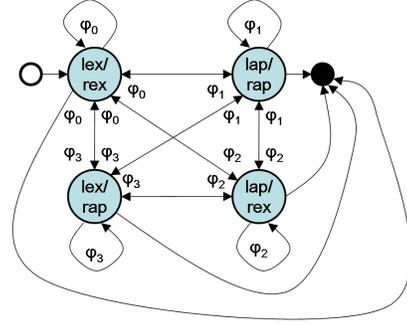


Fig. 2. State machine for adaptive join control

4 Experimental Evaluation

To evaluate our approach, we have used a variety of test *patterns of data perturbation*, produced using a data generator, for example a uniform distribution of variants across the length of an input, 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. These patterns are designed to simulate various real-life configurations, where batches of data from different sources are collated possibly at different times. The specific patterns used in the evaluation are shown in Fig. 3.

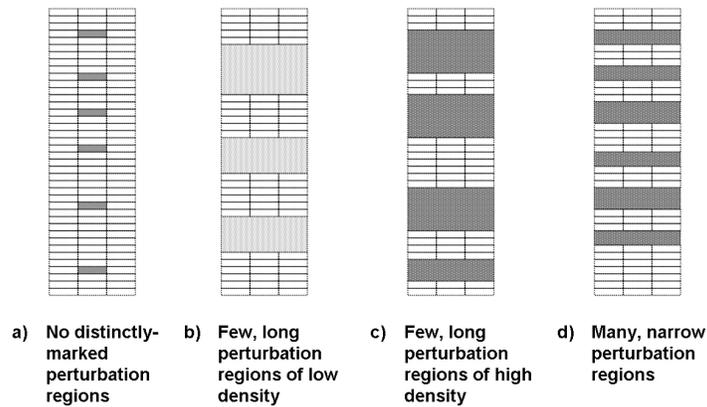


Fig. 3. Perturbation patterns

The suite of parameters that are used by the assessor makes for a potentially large space of configurations. The results presented here below refer to the best possible configuration, determined experimentally, for each of the patterns. The performance metrics used to evaluate the approach are aimed at measuring the relative gain with respect to a baseline. Regarding result completeness we use the number of matched pairs returned by the all-exact join as 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.

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

Regarding cost, we observe that 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, denoted sc_i , plus (ii) the overhead cost due to all the state transitions, denoted tc_i , resulting in the total cost:

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

If we consider the difference between the lowest possible cost c , achieved by using the exact join throughout, and the highest cost C , incurred when the approximate join is used throughout, we obtain the relative cost: $c_{rel} = \frac{C_{abs}-c}{C-c}$.

Fig. 4 shows the overall gain/cost results across a set of eight test cases. These are the best results across a range of parameter configurations, determined experimentally.

The efficiency index $e = \frac{g_{rel}}{c_{rel}}$ i.e., the gain/cost ratio, reported under each column is not just always positive, but it is also contained within a small interval, regardless of the specific pattern used for the test case. This is an indication that the method is fairly robust to variations in the specific perturbation patterns.

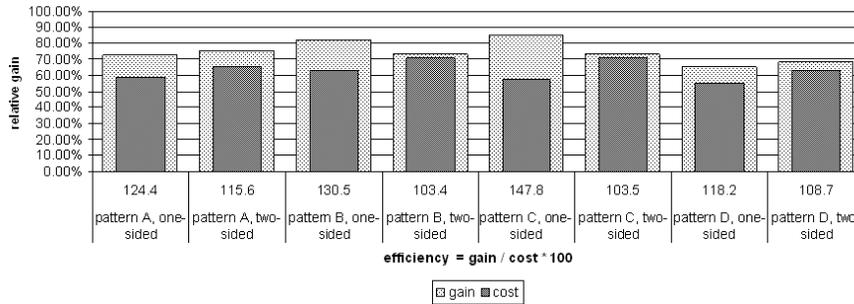


Fig. 4. Gain and Cost across all Test Cases

5 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. S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.
2. S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *ICDE*, pages 865–876, 2005.
3. A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
4. A. K. Elmagarmid, P. G. Ipeirotis, and V.S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, Jan 2007.
5. K. Eurviriyankul, A. A. A. Fernandes, and N. W. Paton. A foundation for the replacement of pipelined physical join operators in adaptive query processing. In *EDBT Workshops*, pages 589–600, 2006.
6. Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to source properties in processing data integration queries. In Weikum et al. [10], pages 395–406.
7. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.
8. R. Lengu, P. Missier, A. A. A. Fernandes, G. Guerrini, and M. Mesiti. Time-completeness trade-offs in record linkage using adaptive query processing. In *Procs. EDBT*, St. Petersburg, Russia, March 2009.
9. Roald Lengu, Giobanna Guerrini, and Marco Mesiti. An adaptive join technique for result completeness. Technical Report DISI-TR-08-03, DISI, Università di Genova, Via Dodecaneso, Genova, IT, 2008. <http://www.disi.unige.it/>.
10. Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 2004.
11. Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77. IEEE Computer Society, 1991.