Matching and Evaluation of Disjunctive Predicates for Data Stream Sharing*

Richard Kuntschke
Technische Universität München
Munich, Germany
richard.kuntschke@in.tum.de

Alfons Kemper
Technische Universität München
Munich, Germany
alfons.kemper@in.tum.de

ABSTRACT

New optimization techniques, e. g., in data stream management systems (DSMSs), make the treatment of disjunctive predicates a necessity. In this paper, we introduce and compare methods for matching and evaluating disjunctive predicates.

Categories and Subject Descriptors: H.3 [**Information Systems**]: Information Storage and Retrieval; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*

General Terms: Algorithms, Performance

Keywords: Disjunctive predicates, predicate matching, predicate evaluation, data stream sharing

1. INTRODUCTION

Except for a few publications which have dealt with the issue in the database field, disjunctive predicates, which are known to be complex to handle, have largely been neglected in the context of query optimization for traditional database management systems (DBMSs). Instead, query optimization generally limits itself to considering conjunctive query predicates since well-known ways for efficiently managing such predicates exist. The main argument for justifying the neglect of disjunctive predicates has been that such predicates do not occur often in practice. While this argument might be true for traditional database systems and applications, it is not correct for new and evolving applications and optimization techniques, e. g., in the domains of semantic caching and data stream management systems (DSMSs). One such optimization technique is *data stream sharing* [1] which reuses data streams in a distributed DSMS for satisfying multiple similar continuous queries, thus reducing network traffic and computational load.

In this paper, we introduce and compare methods for matching and evaluating disjunctive predicates. Predicates in our context are disjunctions of conjunctive predicates. Each conjunctive predicate is a conjunction of atomic predicates. Atomic predicates are of the form $v\ \theta\ c$, where v is a variable, c is a constant, and $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

2. PREDICATE MATCHING

Predicate matching is the problem of deciding whether a predicate implies another and, if this is not the case, how the other predicate can be altered in order for the implication to be valid. More formally, given two predicates p and p', matching p' with p returns (true, p), if $p' \Rightarrow p$, and $(\text{false}, \bar{p})$, where $\bar{p}$ is a relaxed version of p such that $p' \Rightarrow \bar{p}$ (and of course also $p \Rightarrow \bar{p}$), otherwise.

*This research is supported by the German Federal Ministry of Education and Research (BMBF) within the D-Grid initiative under contract 01AK804F and by Microsoft Research Cambridge (MSRC) under contract 2005-041.

CIKM'06, November 5–11, 2006, Arlington, Virginia, USA.
ACM 1-59593-433-2/06/0011.

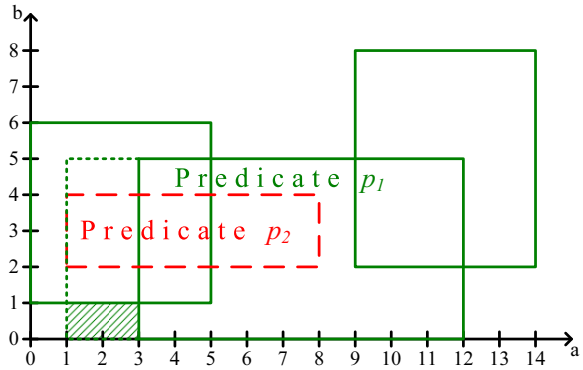

Figure 1: Visualization of example predicates

A graphical representation of two example predicates p_1: $((a \geq 3) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5)) \vee ((a \geq 9) \wedge (a \leq 14) \wedge (b \geq 2) \wedge (b \leq 8)) \vee ((a \geq 0) \wedge (a \leq 5) \wedge (b \geq 1) \wedge (b \leq 6))$ (solid boxes) and p_2: $((a \geq 1) \wedge (a \leq 8) \wedge (b \geq 2) \wedge (b \leq 4))$ (dashed box) is shown in Figure 1.

2.1 Quick Check

Before the actual predicate matching algorithms, we first introduce a simple *quick check* (QC) algorithm that can be combined with each of the matching algorithms. It tests for a conjunctive subpredicate c', whether it implies at least one of the conjunctive subpredicates c of a given predicate p. The implication check for conjunctive predicates can easily be done by checking the bounds of c' for containment in the intervals defined by the atomic predicates in c for all dimensions. If the quick check returns true, nothing more remains to be done for the tested subpredicate c' because it is clear that it already matches predicate p as is.

Concerning our example, comparing the only conjunctive subpredicate of p_2 to each conjunctive subpredicate of p_1 obviously yields no match, i. e., the quick check returns false. This is due to the fact that none of the tested implications is valid which can easily be seen from Figure 1. The dashed box of p_2 is not completely contained in any one of the three solid boxes of p_1.

2.2 Heuristics with Simple Relaxation

The easiest way to perform predicate matching is to completely skip the predicate implication checking and go directly to the relaxation part. This is the idea of the *heuristics with simple relaxation* (HSR). When matching a predicate p' with a predicate p, all conjunctive subpredicates of p' are disjunctively added to p. Since this solution does not perform any implication checking at all, it will miss matches already present in the original predicates and perform unnecessary relaxations in general.

The situation can be improved by combining the approach with the quick check algorithm of Section 2.1. The matching problem for disjunctive predicates is thereby basically reduced to the implication problem for conjunctive predicates. In this solution, two nested loops compare each conjunctive subpredicate of p' to each conjunctive subpredicate of p, checking for implication. If, for

each subpredicate of p', a matching subpredicate in p is found, the matching succeeds, else it fails. Obviously, this approach might fail even though p' and p do match. In the running example, the only subpredicate of predicate p_2 does not match any of the three subpredicates of predicate p_1 directly. However, it matches the whole predicate p_1, as can be seen from Figure 1, which this algorithm would not realize. Therefore, a mismatch would be reported although the predicates actually do match. Predicate relaxation in the case of a mismatch is simply done by adding the concerned subpredicate of p' to p using a disjunction.

The advantages of the HSR algorithm are that it is fast and easy to implement. The disadvantages of the approach obviously are that it generally misses matches—actually all matches if it is used without the quick check—and that it can therefore cause unnecessary predicate relaxations which affects the performance of future predicate matching and evaluation processes.

2.3 Heuristics with Complex Relaxation

The *heuristics with complex relaxation* (HCR) avoids the increase in the number of subpredicates in p induced by HSR at the expense of potentially producing only approximate results. For each conjunctive subpredicate in p', HCR relaxes one of the conjunctive subpredicates in p in order for it to match the subpredicate of p' if no direct match between subpredicates has been found. In the DSMS scenario, relaxing a subpredicate means employing a less restrictive filter on the corresponding data stream, therefore increasing network traffic. Thus, the subpredicate of p which needs the least amount of relaxation in order to match p' should be relaxed. The situation is illustrated in Figure 1 for our running example. The dotted line marks the relaxation of a subpredicate of p_1. In general, this kind of relaxation causes the data stream to contain unnecessary data. However, in the example, parts of this data are already covered by another conjunctive subpredicate of p_1 as can be seen from Figure 1. Therefore, additional unnecessary network traffic is only caused by the inclusion of the hatched area in Figure 1 in this specific case. HCR, like HSR, can be combined with the quick check algorithm of Section 2.1 to detect obvious matches before starting the more complex relaxation algorithm.

Similar to HSR, HCR is relatively fast and easy to implement. Furthermore, it does not introduce any additional disjunctions in the stream predicate. The disadvantages are that the approach still misses matches and therefore performs unnecessary predicate relaxations in general. Additionally, HCR, in contrast to HSR, can lead to the inclusion of unneeded parts of the data space in the relaxed predicate and therefore cause unnecessary network traffic through false drops. This necessitates additional filtering to obtain the exact result if approximate results are not acceptable.

2.4 Exact Matching

The *exact matching* (EM) algorithm is a split algorithm that always correctly detects a match of p' with p. It does not miss matches like the heuristics above nor does it report false matches. Predicate p' is split along its dimensions according to the boundaries of the overlapping intervals of p. Only if, at the end of the matching process, all parts of p' have been successfully matched, a match is reported. Otherwise, p is relaxed. The example of Figure 1 illustrates the case of a complete match between predicates. In case of a mismatch, we relax p by disjunctively adding the unmatched subpredicates of p'. Note that the HSR and EM algorithms, as opposed to the HCR algorithm, never add unnecessary parts of the data space to p during relaxation. Again, the quick check presented in Section 2.1 can be executed in combination with the EM approach to check for matching subpredicates in advance before starting the more complex relaxation algorithm.

The exact solution has the advantage of determining matches between predicates in an exact way, i. e., producing no false matches and—as opposed to the heuristics—finding all existing matches. Therefore, the approach does not cause any unnecessary predicate relaxations. Also, it can exactly identify the non-matching parts of p'. The major disadvantage of the exact solution is its high algorithmic complexity, which is exponential in the number of subpredicates in the predicates to be matched. This might slow down the optimization process considerably for predicates with many disjunctions and makes the algorithm inapplicable for larger problem sizes. In such cases, the heuristics have to be used.

3. PREDICATE EVALUATION

Apart from predicate matching, efficient predicate evaluation is of major importance in a DSMS. Predicate evaluation is the problem of deciding whether a data item satisfies a predicate or not. More formally, given a predicate p and a data item i, evaluating p against i returns true, if, for all dimensions referenced in p, the value of i in the corresponding dimension lies within the interval defined for that dimension in p. The goal is to evaluate a given predicate against as many data items per time unit as possible, i. e., achieve a high throughput. In the following, two approaches for predicate evaluation are presented.

3.1 Standard Evaluation

We use the term *standard evaluation* (SE) to denote a simple sequential scan. It evaluates a given predicate p against a given data item i by iterating over the conjunctive subpredicates c of p and testing for each dimension, whether the value of i in that dimension lies within the interval defined for the same dimension in c. As soon as a subpredicate containing the values of i in each dimension is found, the algorithm terminates and returns true. Only if, after inspecting all conjunctive subpredicates c of p, no subpredicate containing i could be found, it returns false.

3.2 Index-based Evaluation

Considering the facts that the exact matching algorithm is only applicable for small problem sizes and the approximate results of the HCR algorithm are often not desirable, a switch to the HSR algorithm for larger problem sizes, i. e., larger numbers of dimensions and subpredicates, seems necessary in many cases. Since this algorithm—with as well as without quick check—can introduce a considerable number of additional disjunctions in predicates, the standard evaluation algorithm above will quickly become inefficient. Therefore, an optimized predicate evaluation strategy that better handles large numbers of subpredicates is needed.

We can optimize predicate evaluation using multi-dimensional indexing. We use the term *index-based evaluation* (IE) to denote the evaluation algorithm with index support. It differs from standard evaluation in that it uses a multi-dimensional index structure to represent the predicate. To evaluate the predicate against a data item, the algorithm simply executes the containment method of the index with the data item as its only parameter. The evaluation is then performed completely by the index, returning true if the predicate covers the data item and false otherwise. Note that the matching efficiency of the QC and EM algorithms of Section 2 can also benefit from multi-dimensional indexing.

4. CONCLUSION

In this paper, we have presented various methods for matching and evaluating interval-based disjunctive predicates. Matching involves deciding whether a predicate implies another and, if this is not the case, how the other predicate can be altered in order for the implication to be valid. More details and an extensive performance evaluation can be found in the full paper [2].

An interesting issue for future work is to design a specialized index structure that specifically fits the needs of indexing disjunctive predicates. In this course, the functionality of predicate matching could be fully or partially incorporated into the index itself.

5. REFERENCES

[1] R. Kuntschke and A. Kemper. Data Stream Sharing. In *Proc. of the Intl. Workshop on Pervasive Information Management*, Munich, Germany, Mar. 2006.

[2] R. Kuntschke and A. Kemper. Matching and Evaluation of Disjunctive Predicates for Data Stream Sharing. Technical Report TUM-I0615, Technische Universität München, 2006.