have developed strategies for estimating selectivity based on suffix trees by drawing on our intuition of the close relationship between prediction and data compression [Kri]. We have shown how to efficiently build a small structure that fits in the metadata in the runstats phase, and how to match query patterns against such a structure. Experimental evidence based on data from the TPC-D benchmark suggests that our techniques hold great promise for effective use in real database systems. Experiments against real customer data were subsequently done validating the conclusions of this paper [Wan]; more work is ongoing and preliminary results are encouraging [Wan]. It will be interesting to see if better strategies for matching against our reduced tree structure can be developed.

An important conclusion is that selectivity estimation for text is possible by applying the principles of data compression. If this observation holds true for audio, video, and image databases, it would benefit database management systems immensely.

References

[ASW] M. M. Astrahan, M. Schkolnick, and K. Y. Whang, "Approximating the Number of Unique Values of an Attribute Without Sorting," *Inf. Sys.* 12 (1987), 11–15.

[BGI] G. Bhargava, P. Goel, and B. R. Iyer, "Hypergraph Based Reorderings of Outer Join Queries with Complex Predicates," *Proc. of the 1995 ACM SIGMOD Conference*, 304–315.

[BDG] S. Brin, J. Davis, and H. Garcia-Molina, "Copy Detection Mechanisms for Digital Documents," *Proc. of the 1995 ACM SIGMOD Conference.*

[BuB] S. Bunton and G. Borriello, "Practical Dictionary Management for Hardware Data Compression," Dept. of Comp. Sci., Univ. of Washington, FR-35, 1991.

[CKV] K. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching via Data Compression," *Proc. of the 1993 ACM SIGMOD Conference*, 257–266.

[Dat] C. Date, *An Introduction to Database Systems*, Addison-Wesley, 1981.

[FiG] E. R. Fiala and D. H. Greene, "Data Compression with Finite Windows," *Comm. of the ACM* 32 (April 1989), 490–505.

[HaSa] P. Haas and A. Swami, "Sequential Sampling Procedures for Query Size Estimation," *Proc. of the 1992 ACM SIGMOD Conference*, 341–350.

[HaSb] P. J. Haas and A. N. Swami, "Sampling-Based Selectivity for Joins Using Augmented Frequent Value Statistics," *Proc. of the 11th Intl. Conf. on Data Engg.* (March 1995).

[HeS] M. A. Hernandez and S. J. Stolfo, "The Merge/Purge Problem for Large Databases," *Proc. of the 1995 ACM SIGMOD Conference*, 127–138.

[IoP] Y. E. Ioannidis and V. Poosala, "Balancing Histogram Optimality and Practicality for Query Result Size Estimation ," *Proc. of the 1995 ACM SIGMOD Conference.*

[Knu] D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[Kri] P. Krishnan, "Online Prediction Algorithms for Databases and Operating Systems," Brown Univ. Ph.D. thesis, May, 1995, also available as Brown Univ. technical report CS–95–24.

[LNS] R. Lipton, J. Naughton, and D. Schneider, "Practical Selectivity Estimation through Adaptive Sampling," *Proc. of the 1990 ACM SIGMOD Conference*, 1–11.

[LiN] R. J. Lipton and J. F. Naughton, "Query Size Estimation by Adaptive Sampling," *J. Comput. Sys. Sci.* 51 (August 1995), 18–25.

[McC] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *J. of the ACM* 23 (1976), 262–272.

[Mor] D. R. Morrison, "PATRICIA–A Practical Algorithm to Retrieve Information Coded in Alphanumeric," *J. of the ACM* 15 (1968), 514–534.

[SAC] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," *Proc. of the 1979 ACM SIGMOD Conf.*, 23–34.

[TPC] Transaction Processing Performance Council, TPC, "TPC Benchmark^{TM} D (Decision Support), Working Draft 6.0," 1994, F. Raab (ed.).

[Vap] V. N. Vapnik, *Estimation of Dependencies based on Empirical Data*, Springer Verlag, 1982.

[WCM] J. T. Wang, G. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang, "Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results," *Proc. of the 1994 ACM SIGMOD Conference* (May 1994).

[Wan] M. Wang, Duke Univ., personal communication.

[Wei] P. Weiner, "Linear Pattern Matching Algorithms," *Proc. of the IEEE 14th Annual Symp. on Switching and Automata Theory* (October, 1973), 1–11.

[WVT] K.Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A Linear-Time Probabilistic Counting Algorithm for Database Applications," *ACM Trans. on Database Sys.* 15 (June 1990), 208–229.

[ZiLa] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. on Inf. Theory* 23 (May 1977).

[ZiLb] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Trans. on Inf. Theory* 24 (September 1978), 530–536.

Pattern	I_1	I'_1	I_2	I'_2	I_3	I'_3	CE_1	CE_2	CE_3	DE_1	DE_2	RS
positive single	64	65	28	28	57	57	49	100	82	85	75	34
positive double	60	63	0	0	41	41	34	0	53	22	11	0
negative	46	39	21	0	21	14	90	31	42	31	16	100

Table 1: Table of grades for the different algorithms. A grade of x means that the algorithm did "well" (as defined in Section 7.4) for $x\%$ of the queries. The right-most column presents results for the random sampling strategy from Section 8. Total data structure memory usage in all cases is 389 bytes.

lenient with negative patterns, since we expect them to occur less frequently. We are also less critical about under-estimating as opposed to over-estimating.

Table 1 presents the grades of the different algorithms for positive single patterns, positive double patterns, and negative patterns, where a negative pattern is obtained by introducing two errors into a single positive pattern. We conclude from Table 1 and our discussion from Sections 7.1–7.3 that strategies I_1, I'_1, CE_1, CE_3, and DE_1 hold promise for good estimation of alphanumeric selectivity.

8 A "Strawman" Random Sampling Scheme (RS)

It is instructive to compare the results presented in Section 7 against a simple random sampling strategy motivated by work in learning theory [Vap]. The comparisons are especially interesting given that we know of no prior work in estimating alphanumeric selectivity. Sampling is also used for estimating numerical selectivity [HaSa, HaSb, IoP, LNS, LiN].

In sampling, we take a random sample of the strings in the column of the database and store as many as possible in the metadata, limited by the size of the metadata allowed for the statistics. When a query pattern is presented, the pattern is matched against the strings stored in the metadata, and the selectivity is estimated in the obvious way: If there are ℓ strings in the metadata and q of them match the query pattern, the selectivity is estimated to be q/ℓ. This approach will always correctly estimate the selectivity to be 0 for negative patterns.

In the right-most column of Table 1, we present the grades for the random sampling scheme. (12 strings were stored using the 389 bytes of memory available in the metadata.) We observe that our recommended matching strategies from Section 7.4 outperform the random sampling strategy for positive single patterns and especially for positive double patterns. Strategy CE_1 does almost as well as the random sampling strategy for negative patterns, and does better for both positive single and positive double patterns.

The random sampling strategy presented above is in a basic or "strawman" form. Modifications to this basic strategy, such as a hybrid strategy incorporating the approaches in Section 4, are currently being investigated and will be reported in a future paper [Wan].

9 Extensions

In this section we describe certain extensions to our method for estimating selectivity. For brevity, we omit details from this abstract.

Non-Unit Patterns. Given a general pattern that begins with a wildcard (e.g., "$*\alpha*\beta*$"), we can break it up into sub-patterns that are each unit patterns (e.g., "$*\alpha*$" and "$*\beta*$"), separately estimate selectivity for each of the sub-patterns, and put them together using ideas similar to the ones presented in Section 4. For patterns that do not begin with a "$*$", the situation is somewhat different; we can, however, keep another count called *count′* with each node of the tree (or an approximate count, or an approximation of *count′*/*count*), which specifies the number of rows of the column that will match a query that does not start with a wildcard.

Optimization With Extra Knowledge. It is important to note that the methods we have presented for creating the reduced tree, the strategies for matching, and our experimental results do not assume any knowledge of the distribution or type of data in the column of the database, and do not use specifics of how entries are constructed in the emerging TPC-D benchmark. In practice, extra information, if available, can be used. For example, if we know that a character x (e.g., the blank character) is very likely to occur in the rows of the database, we could presumably prune out the whole subtree below character x while generating the reduced tree τ without losing much information. (The estimation method would be adjusted accordingly.) Average size of match patterns can also be used to tune the information in the metadata. Further, if statistics are gathered when data is re-organized, there may be additional opportunity for better estimation of selectivity.

10 Conclusions

In this paper, we have performed the first known study of estimating alphanumeric selectivity. We

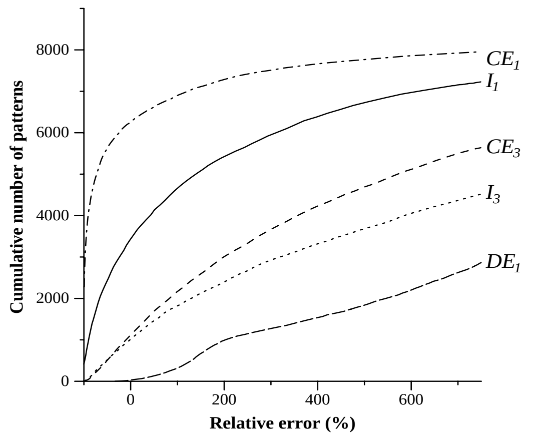


Figure 8: Graph of the best performing strategies for positive double patterns. The graphs are cropped at $x = 750$. The number of different query patterns is $92 \times 91 = 8372$.

itive single patterns. The selectivity of the double patterns are very small (approximately 0.0005). A larger relative error in this case corresponds to a much smaller absolute error.

We observe from Figure 8 that the relative performance of the strategies for positive double patterns is $I_1 > CE_3 > I_3 > CE_1 > DE_1$, where ">" implies "is better than." The strategies that perform well for the positive single patterns, namely, strategies I_1 and CE_3, also perform well for positive double patterns. In addition, the trends shown by the strategies for the positive single patterns also hold for positive double patterns. The notable exception is strategy CE_2; all the double patterns have a relative error greater than 2000% with strategy CE_2. This is not surprising since strategy CE_2 does not use the parsing of the pattern effectively. Interestingly, strategies I_3 and CE_1 perform reasonably well and strategy I_1' performs slightly better than strategy I_1 for positive double patterns.

7.3 Negative Patterns

In this section, we present our results for negative patterns, where each negative pattern was obtained by introducing two errors into each positive single pattern. The relative performance of the strategies are qualitatively similar to the situation when one or three errors are introduced in the single patterns; the selectivity is closer to zero (i.e., better) when the pattern is formed by introducing more errors.

In the case of negative patterns, we want the the curve to lie as close to the y-axis as possible. (See Figure 9.) We plot curves for the three strategies that perform well for single and double positive patterns, i.e., strategies I_1, CE_3, and DE_1. We observe that these strategies perform well for negative patterns also, with strategy I_1 doing better than strategies CE_3 and

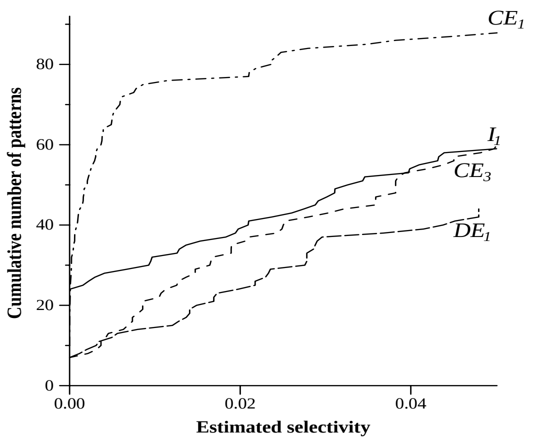


Figure 9: Graph of the best performing strategies for negative patterns obtained by introducing two errors.

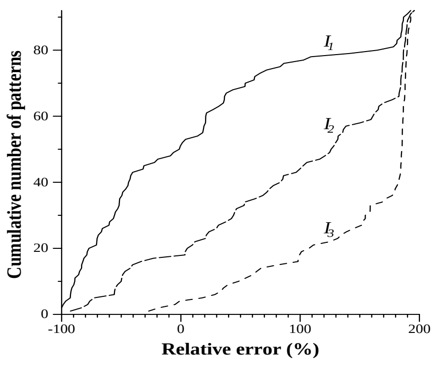
DE_1. For negative patterns, strategy I_1 performs a little better than strategy I_1'. We also plot the curve for one other strategy, CE_1, that performs particularly well for negative patterns.

7.4 Comparing the Strategies Globally

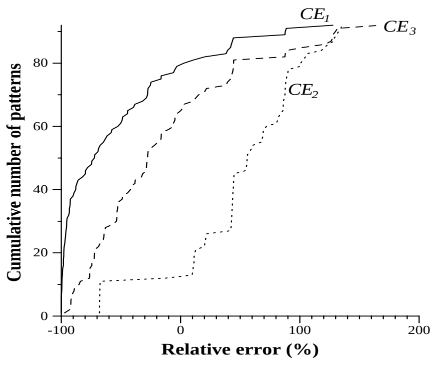
We have observed that some strategies perform particularly well for specific types of patterns (e.g., strategy CE_1 for negative patterns, and strategy CE_2 for positive single patterns). However, we want strategies that are consistently good over all types of patterns: positive single patterns, positive double patterns, and negative patterns. We have observed three strategies that consistently perform well for all types of patterns, namely strategies I_1 (or I_1'), CE_3, and DE_1. In this section, we develop a quantitative measure to evaluate the strategies.

Our goal is to assign a *grade* to each strategy based on certain intuitive notions we have of how much error is acceptable. Given a measure of acceptable error, we define the grade of the strategy as the percentage of patterns that fall within the acceptable error range. The acceptable error range would depend on the type of pattern. Other measures that take into account the entire distribution (i.e., more continuous measures that study degrees of acceptance) can also be considered.

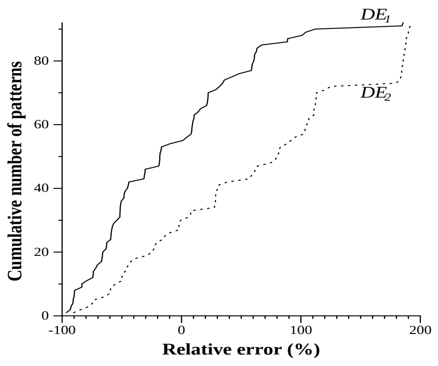
For our study, we define a relative error between -75% and $+150\%$ as acceptable for positive single patterns, a relative error between -75% and 500% as acceptable for positive double patterns, and an absolute error, where the estimated selectivity is less than 0.025, as acceptable for negative patterns. The logic behind these definitions is based on what we expect will be the action of the query optimizer with respect to the estimations, and is based on the real value of selectivity. For example, we allow a higher relative error as acceptable for positive double patterns, since this corresponds to a small absolute error. We are more



(a) Independence-based



(b) Child Estimation-based



(c) Depth Estimation-based

Figure 6: Performance of the strategies for positive single patterns. The number of different query patterns is 92.

prune probability sufficiently larger than the expected selectivity of the base patterns. We therefore report our results in this section for when the reduced tree holds 100 nodes. We would like to emphasize that for the complexity of the problem, having a larger metadata, if possible, would help significantly.

Evaluation Metric. We first present our results for matching positive single patterns in Section 7.1 and our results for matching positive double patterns in Section 7.2. For a positive pattern σ we measure the *relative error*, i.e., $(est(\sigma) - sel(\sigma))/sel(\sigma)$. Notice that the relative error can never be less than -100%. We plot the cumulative number of patterns along the y axis against the relative error on the x axis; a point (x, y) on the graph implies that y positive single patterns have relative error $\leq x\%$. To find out if a strategy is acceptable, we can draw vertical lines at the maximum and minimum acceptable error; the difference of the y values of a curve where these lines intersect it will give us the number of patterns with acceptable error. *In particular, a "good" strategy has a* steeper slope *(not a larger value, necessarily) around $x = 0$.*

We then look at the case of negative patterns in Section 7.3. In this case $sel(\sigma) = 0$. We therefore measure the *absolute error*; i.e., $est(\sigma)$, which is proportional to the number of rows of the column R that match the pattern. *In this case, a "good" strategy has a large y value close to $x = 0$.* In Section 7.4, we put together all our observations to evaluate the estimation strategies by defining a new comparison metric.

7.1 Positive Single Patterns

Figure 6a shows the performance of the independence-based strategies. There is little difference in performance between the "primed" (e.g., I'_1) and the corresponding "non-primed" (e.g., I_1) strategies; we therefore omit the lines for the "primed" strategies from the graph. We observe that amongst the independence-based strategies, I_1 (and I'_1) perform best. Strategies I_2 and I_3 tend to over-estimate selectivity, with strategy I_3 being a little more accurate than strategy I_2. From Figure 6b, we observe that the child estimation-based strategies perform well; strategy CE_2 has a particularly sharp slope around $x = 0$. From Figure 6c, we observe that strategies DE_1 and DE_2 perform well, with strategy DE_1 having a slightly larger slope around $x = 0$.

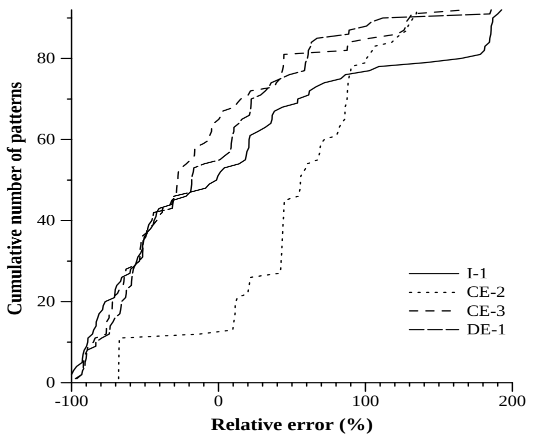


Figure 7: Graph of the best performing strategies for positive single patterns. The number of different query patterns is 92.

Figure 7 plots the best strategies identified above. Strategy CE_2 stands out as being particularly good; strategies CE_3 and DE_1 are slightly better than strategy I_1. (Strategies I_3, I'_3 and CE_1 are not much away from I_1 in performance; their curves are not plotted in Figure 7.)

7.2 Positive Double Patterns

Figure 8 plots for positive double patterns the performance of strategies that were identified as best for pos-

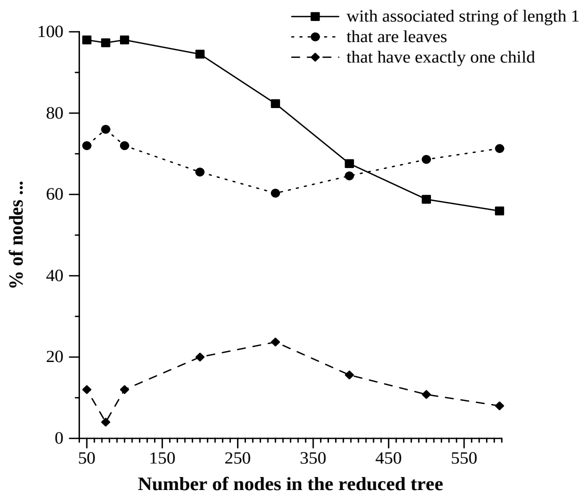


Figure 4: Various statistics related to the runstats phase as a function of the number of nodes in the reduced tree.

way the column is built, each positive single pattern is expected to have a selectivity of $5/92 = 5.4\%$, and each positive double pattern is expected to have a selectivity of 0.048%. (That is, a positive single pattern appears in roughly $200\text{K} \times 5/92 = 10869$ rows.)

In Section 7, we present the results of using our matching strategies from Section 4 for matching positive and negative patterns against our reduced tree. It is important to keep in mind the distinction between positive and negative patterns: positive patterns are present in the column and have a positive selectivity, while negative patterns are not present in the column, and have a selectivity of zero.

6 Results for the Runstats Phase

In stage 1 of the runstats phase, we took each of the 200K patterns in the P_NAME column of the database and inserted them into a suffix tree T. In stage 2, we pruned the tree T to get the reduced tree τ. In this section, by "top nodes" we mean nodes with the highest value of *count*.

Size of the Tree in Stage 1. We varied the maximum number of nodes M retained in the tree in stage 1 from between 300K to 450K. (See Section 3.1.1.) There was little difference in the final reduced trees for these values of M; the value of *count* at a few nodes differed by 1.

Statistics Dealing With Tree Reduction. (See Figure 4.) The solid line shows that most of the strings associated with the top nodes are of length 1. This verifies our intuition from Section 3.2.1 that nodes with high count have strings of small length associated with them. The dotted line in Figure 4 shows that a large percentage of the nodes in the reduced tree are leaves. (Recall that we do not need to store a child pointer with leaf nodes.) The dashed line shows the percentage of nodes that will be present as "strands." Interestingly, the dotted and dashed curves seem to be mirror images of each other.

With $M = 450\text{K}$ in stage 1 of the runstats phase, the maximum depth of the tree is just 15. This implies a small table for the depth vs. count distribution if strategy DE_* from Section 4.3 were to be used.

Reduced Tree Size. We calculated that we could store the top 100 nodes of the reduced tree using 389 bytes, and the top 200 nodes using 1000 bytes. For brevity, we omit details of the breakup of storage use.

Size of the Reduced Tree versus Prune Count. As we have seen in Section 4, the error of our matching strategies in the query optimization phase is closely related to the prune count. From Figure 5, we can see the steep jump in the prune count as the number of nodes decreases.

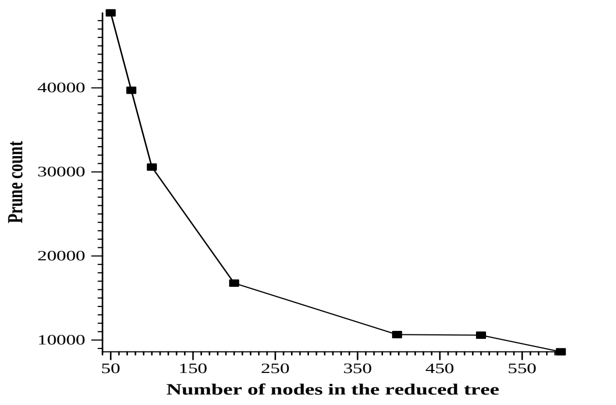


Figure 5: Variation of prune count with the number of nodes retained in the reduced tree.

7 Results for the Query Optimization Phase

From Section 6, we observe that if we are allowed between 0.5–1 Kbytes to store our reduced tree, we can store from between 100–200 nodes. From Figure 5 and the observation at the end of Section 5 on how many rows a positive single pattern appears in, it is obvious that the errors will be minimal when we use a reduced tree with 200 nodes. In particular, with a 200 node reduced tree, for positive single patterns, a strategy that returns the prune probability when a pattern does not match will estimate the selectivity to be $16\text{K}/200\text{K} = 0.08$, while the true selectivity is approximately 0.05; the error is therefore small.

To really test our estimation strategies, and for us to be confident about their validity in environments outside the evolving TPC-D benchmark, it is important that we test them against a reduced tree with a of a node versus $av\_count(d)$, the average value of *count* at depth d while reducing tree T to get τ.

For our running example, the depth vs. count distribution for the tree T from which the reduced tree τ in Figure 3 was derived is $\{(\text{depth } d,\ av\_count(d)\}$:

$$\{(0,1000),(1,500),(2,275),(3,250)\} \qquad (3)$$

We denote the depth of node z by $depth(z)$.

Strategies DE_1, DE_2: The depth estimation strategies try to estimate the depth of pattern σ in T as a weighted average of the $depth(z(\sigma(i)))$'s, with the weights decreasing with increasing i. Intuitively, we want the ith weight to capture the ratio of the quantity depth of pattern $\sigma(0)\sigma(1)\cdots\sigma(i)$ in T minus the depth of pattern $\sigma(0)\sigma(1)\cdots\sigma(i-1)$ in T to $depth(z(\sigma(i)))$.

Strategy DE_1 weights the depth of the ith subpattern, $depth(z(\sigma(i)))$, by $1/(i+1)$, and strategy DE_2 weights $depth(z(\sigma(i)))$ by $1/(2i+1)$. For our example, the depth of each of $\sigma(0)$, $\sigma(1)$, and $\sigma(2)$ in the reduced tree is 1. Strategy DE_1 estimates the depth of pattern σ as $1\cdot 1 + 1/2\cdot 1 + 1/3\cdot 1 = 1.83$. Using (3), it estimates the selectivity of pattern σ as $(av\_count(1) + 0.83\cdot(av\_count(2) - av\_count(1)))/|R| = 0.313$.

5 Methodology for the Experiments

We experimentally evaluated our strategies from Sections 3 and 4 for the runstats and query optimization phases. There is an emerging industry standard Transaction Processing Council (TPC) decision support benchmark, known as the TPC-D benchmark [TPC], that involves predicates such as the *like* predicate. We studied our techniques in the context of this benchmark[6].

In particular, we used the *dbgen* program, which is part of the evolving TPC-D benchmark, with a scale factor of one to generate a database. (A scale factor of one corresponds to a database of size 1 Gbyte.) From amongst the columns that the benchmark suggests for use with queries involving the *like* predicate, we chose the P_NAME column of the PARTS table to test our techniques. The P_NAME column is "harder" in that it is expected to have a large number of distinct entries; we expect this column to be "typical" in that the strings in this column are not random alphanumeric strings. For a scale factor of one, the P_NAME column is expected to have 200K rows.

The benchmark specifies that the P_NAME column is populated as follows [TPC]: Each row of the P_NAME column is obtained by choosing five distinct base color patterns at random from a set of 92 patterns ("`green`" is a base pattern, for example), and concatenating these five patterns separating any two patterns with a blank character. It is important to note that we *have not* used specific information about how the column is populated to optimize our techniques; we discuss more about this in Section 9.

As mentioned in Section 1, the space available in the metadata descriptors for any one column of the database is limited. For numeric or measured values, this is determined by known methods (e.g., histograms). A method for alphanumeric selectivity requiring comparable space is desirable. In a database of a million rows, typically each of 100 bytes or more, 0.5–1K would be acceptable, given the large size of the tables. As an initial design point, our work is targeted to these estimates.

In Section 6, we present our experimental observations and validations of our techniques from Section 3 used in the runstats phase.

In the query optimization phase, we need to match query patterns against our reduced tree. The benchmark [TPC] does not provide strict guidelines on how to generate query patterns; it suggests that unit patterns formed from the base patterns be used. We do a more extensive study. We first matched each of the 92 base patterns against our reduced tree and determined the error distribution. (Matching a base pattern σ is equivalent to a *like* predicate with the query pattern "$*\sigma*$".) We call this the *positive single pattern* study, since each pattern exists in the column and the pattern is a base, or single, pattern. We also considered the $92\times 91 = 8372$ *double patterns* obtained by concatenating each of the unit patterns with another, separating these two patterns by a blank. These patterns also exist in the column, and we call this the *positive double pattern* study. Most patterns used with the *like* predicate are *positive patterns*, i.e., patterns that are present in the database [TPC]; hence, the above two studies are most important.

Although *negative patterns*, i.e., patterns not found in the database, are often not seen in query patterns [TPC], it is important to understand the effect of our matching strategies via-a-vis negative patterns, especially given the issue of unclean data [HeS]. We generated three sets of negative patterns by taking the 92 base patterns and introducing one, two, and three errors in each basic pattern, respectively. These are particularly harsh negative patterns since they differ very little from a corresponding positive pattern. The intuition behind generating these negative patterns is that, even if a user is not malicious, people sometimes make typing mistakes, and there may be no records matching on the *like* predicate.

By simple calculations, we observe that from the

[6]Experiments against real customer data were subsequently done validating the conclusions of this paper [Wan].

wise, $est(\sigma(i))$ is defined to be 0 for strategy I_1, and is defined to be the prune probability for strategy I'_1. The estimated selectivity of pattern σ is $est(\sigma) = \Pi_0^m est(\sigma(i))$. The advantage of this approach is that the approximation is likely to be large for queries with large output, but will almost always be small when the actual query output is small. For our running example, strategy I_1 estimates the selectivity of pattern σ to be $est(\sigma(0)) \cdot est(\sigma(1)) \cdot est(\sigma(2)) = (500/1000) \cdot (750/1000) \cdot (500/1000) = 3/16 = 0.188$.

Strategies I_2, I_3, I'_2, I'_3: This set of approaches is based on the observation that the selectivity of pattern σ is at most the selectivity of suffix σ_i, for all i. Strategies I_2 and I_3 use strategy I_1 as a subroutine to determine $est(I_1, \sigma_i)$. They estimate the selectivity of pattern σ as a weighted average of the estimated selectivities $est(I_1, \sigma_i)$. Intuitively, we would like to weight the estimations of the longer patterns more than the estimations of the shorter patterns. Strategy I_2 uses weights varying linearly with the length of the pattern, weighting $est(I_1, \sigma_i)$ by $|\sigma_i| = |\sigma| - i$, while strategy I_3 uses weights falling exponentially with the length of the patterns, weighting $est(I_1, \sigma_i)$ by $2^{|\sigma|-i}$. Strategies I'_2 and I'_3 are similar to I_2 and I_3, except that they use strategy I'_1 as a subroutine, instead of using I_1 as a subroutine.

4.2 Child Estimation-based Strategies CE_*

The logic behind the child estimation-based strategies is to estimate the number of children that were pruned at a node while building the reduced tree τ from T.

For a node z, let $sum\_child\_counts(z)$ be the sum of the counts of the children of z that are in the reduced tree. We denote by $unaccounted\_count(z)$ an estimate of the sum of the values of $count$ of the pruned children of z. We set $unaccounted\_count(z) = count(z) - sum\_child\_counts(z))$ if $count(z) > sum\_child\_counts(z)$, and equal to p otherwise.

Let us denote by $num\_pruned\_children(z)$ the estimated number of children of node z that were pruned in stage 2 of the runstats phase. Since each pruned child of z had a value of $count$ at most the prune count p, we estimate $num\_pruned\_children(z)$ to be $\max(1, \lceil unaccounted\_count(z)/p \rceil)$. (The estimate for $num\_pruned\_children$ assumes that the values of $count$ for the pruned children were approximately equal.)

The CE_* strategies use the $num\_pruned\_children(z)$ values to estimate the selectivity of pattern σ. Intuitively, we expect that the selectivity of pattern $\sigma(i)\theta$, where θ is some character where pattern $\sigma(i)\theta$ does not match in the reduced tree τ, is closely related to

$$\mathcal{S}(z(\sigma(i))) = \frac{unaccounted\_count(z(\sigma(i)))}{num\_pruned\_children(z(\sigma(i))) \cdot |R|}.$$

Strategy CE_1: Along the lines of strategy I_1, this strategy is based on the premise that the substrings $\sigma(i)$'s might be independent, and uses an approximation of Bayes rule. We explain this strategy via our running example.

Strategy CE_1 estimates the selectivity of $\sigma(0) =$ "`gre`" to be $count(x_1)/|R| = 0.5$. It estimates the selectivity of $\sigma(0)\sigma(1) =$ "`gree`" to be $\mathcal{S}(x_1) = 250/(1 \cdot 1000) = 0.25$. Strategy CE_1 expects that of the rows of R that have the substring "`gre`", an "`e`" follows "`gre`" in 0.25/0.5 or half the rows. Similarly, from node x_4 it estimates that "`e`" has a selectivity of 0.75, and that "`en`" has a selectivity of $\mathcal{S}(x_4) = 750/(3 \cdot 1000) = 0.25$. Hence it expects that of the rows that have the substring "`e`", an "`n`" follows an "`e`" 0.25/0.75 or 1/3rd of the time. By an approximate adaptation of Bayes rule, strategy CE_1 estimates the selectivity of σ to be $0.5 \times (0.25/0.5) \times (0.25/0.75) = 0.0833$.

Strategy CE_2: Like strategies I_2, I_3, this strategy is based on the observation that the selectivity of pattern σ is at most the selectivity of suffix σ_i. For our running example, strategy CE_2 estimates that "`gre`" followed by any string of characters not starting with a "`y`" has a selectivity of $\mathcal{S}(x_1) = 0.25$. For $\sigma_1 =$ "`reen`" it estimates that "`re`" followed by any string of characters has a selectivity of 0.25 (using node x_3). By continuing thus (for strings σ_2, σ_3, and σ_4) it estimates the selectivity of σ as $\min_i\{\mathcal{S}(x_i)\}$, where $i = 1, 3, 4, 5$, giving $est(CE_2, \sigma) = 0.25$.

Strategy CE_3: Strategy CE_3 is a mix of strategies CE_1 and CE_2. Instead of multiplying the estimated selectivities of the different sub-patterns (like strategy CE_1 does), strategy CE_3 tries to "look down" the (pruned portions of) tree τ, and multiplies the *number of estimated children of the different sub-patterns.* For our running example, since $z(\sigma(0)) = x_1$, strategy CE_3 thinks that the $unaccounted\_count$ at node x_1 would have gotten successively divided if we had gone down the tree T. Since $z(\sigma(1)) = x_4$, strategy CE_3 expects that the $unaccounted\_count$ at node x_1 would have gotten divided amongst $num\_pruned\_children(x_1) \times num\_pruned\_children(x_4) = 1 \times 3$ children. It estimates the selectivity of pattern σ as $\mathcal{S}(x_1)/3 = 250/3000 = 0.083$.

4.3 Depth Estimation-based Strategies, DE_*

The depth estimation strategies combine the intuition of the suffix tree with the histogram-based strategies [IoP] for estimating numeric selectivity. The idea is to approximately capture the distribution of the depth d

bilities include hashing of the strings associated with a node.

To pack the maximum number of nodes into the metadata, we can list the nodes in order of decreasing count, and perform a simple calculation to determine the "break point," i.e., the last node that can be squeezed into the metadata. (More details are given in [Kri].)

4 The Query Optimization Phase

In the query optimization phase, a query using the *like* predicate is presented, and we consult the metadata (actually, the reduced tree τ stored in the metadata) to estimate the selectivity of the *like* predicate. Let the query with the like predicate involve the unit pattern "$*\sigma*$". We call the string σ the "string in the unit pattern" or simply *the pattern*; the selectivity of the *like* predicate is the selectivity of pattern σ.

We use the term *prune probability* to denote the value of prune count (defined in Section 3.2) divided by $|R|$. In this section, we will denote the pattern by σ, and the suffix of the pattern σ starting at position i by σ_i; in particular $\sigma = \sigma_0$. We denote the selectivity of pattern σ by $sel(\sigma)$, and the estimated selectivity of pattern σ using strategy X by $est(X, \sigma)$. When X is obvious from context, we use $est(\sigma)$ to denote the estimated selectivity of pattern σ.

In the query optimization phase we *match* pattern σ against the reduced tree τ to get an estimate $est(\sigma)$ which is as close to $sel(\sigma)$ as possible. As described in Section 3, determining whether the pattern σ is in the reduced tree τ is simple. We have a *successful match* of the pattern if we find z, the node of the match, and an *unsuccessful match* otherwise. Clearly, if we have a successful match, $est(\sigma)$ should be equal to $count(z)/|R|$ for any reasonable estimation strategy, since $est(\sigma)$ will then be equal to $sel(\sigma)$, barring the minimal effects of regular pruning in stage 1 of the runstats phase.

It is quite likely that a pattern that would have successfully matched in the tree T (obtained in stage 1 of the runstats phase) will not match in the reduced tree τ. Since we know that the selectivity in the case of an unsuccessful match is at most the prune probability, setting $est(\sigma)$ equal to the prune probability might be reasonable, especially if the prune probability is small. However, the stringent limits on the size of the metadata (and hence on the size of τ) do not guarantee that the prune probability will always be small. We now concentrate on strategies to get a better approximation for the selectivity for the difficult and more likely case when the pattern does not match in the reduced tree τ.

We have three main families of methods for dealing with unsuccessful matches: *independence-based*, *child estimation-based*, and *depth estimation-based* methods. In each of these methods, we deal with *partial matches*

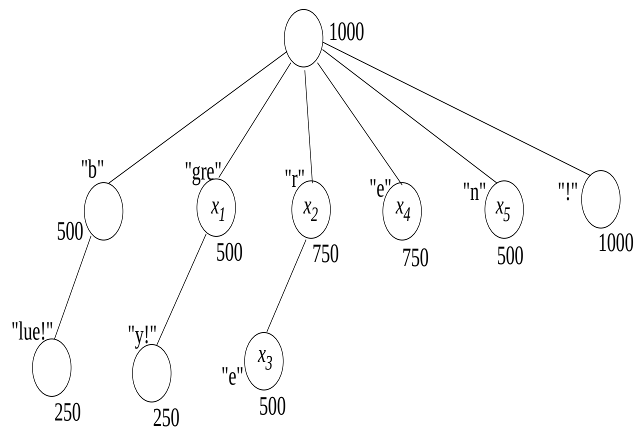


Figure 3: The reduced tree τ obtained by inserting a column R of a hypothetical database having $|R| = 1000$ strings, with 250 of these strings being "`green!`", 250 being "`grey!`", 250 being "`brown!`", and 250 being "`blue!`", and retaining nodes with the maximum 10 counts in the reduced tree. The prune count is 250, and the prune probability is $250/1000 = 0.25$.

of the pattern in the reduced tree τ. The idea behind partial matches is that if a pattern does not match in the reduced tree τ, we can divide the pattern into sub-patterns that match in τ, determine the selectivity of each of these sub-patterns, and put these values of selectivity together in some logical way to get the selectivity of the full pattern.[5] We explain the main idea of our strategies via a running example using the reduced tree τ from Figure 3, and the pattern σ = "`green`".

All our strategies rely on a *greedy parsing* of the pattern σ. Formally, we say that $\sigma(0), \sigma(1), \ldots, \sigma(m)$ is a greedy parsing of σ if $|\sigma(i)| > 0$ for all i, $\sigma(0)\sigma(1)\cdots\sigma(m)$ equals pattern σ, and $\sigma(i)$ is either the maximal match of σ_j in the reduced tree τ, where $j = |\sigma(0)| + \cdots + |\sigma(i-1)|$, or the single character at position j of pattern σ if no non-null prefix of σ_j successfully matches in the reduced tree τ. For our running example, a greedy parsing of σ = "`green`" gives us the sub-patterns $\sigma(0)$ = "`gre`", $\sigma(1)$ = "`e`", and $\sigma(2)$ = "`n`". We denote the node of the match for $\sigma(i)$ by $z(\sigma(i))$.

4.1 Independence-based Strategies, I_*

The independence-based strategies essentially assume that the $\sigma(i)$'s (i.e., the substrings obtained by greedy parsing) are independent of each other.

Strategies I_1, I_1': If $\sigma(i)$ matches in the reduced tree, we define $est(\sigma(i)) = count(z(\sigma(i)))/|R|$. Other-

[5] Although not explicitly stated in the description of the strategies, our algorithms ensure that the estimate of an unmatched pattern is always less than the prune probability.

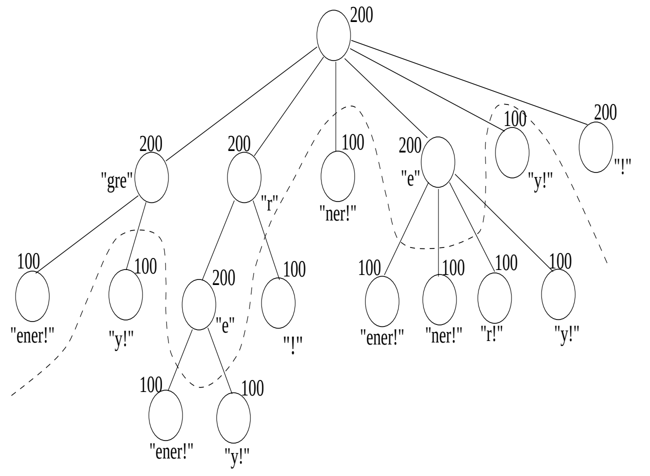


Figure 2: The suffix tree T obtained in stage 1 of the runstats phase by inserting 200 strings into the tree, with 100 of these strings being "`greener!`" and the remaining 100 being "`grey!`". Assume that we can store seven nodes in the metadata. The portion of the tree below the dotted line is pruned away in stage 2 to get the reduced tree τ. The 11-way tie for the seventh node amongst nodes with $count = 100$ is broken arbitrarily. The prune count is 100.

(See Figure 2.) To get tree T, we insert each string σ in column R of the database into a suffix tree[4] T; this involves adding each suffix of string σ to the tree. We also keep a count called $count(z)$ with each node z of the tree of how many strings in column R have as a substring the string corresponding to z. This can be done during the tree creation process by incrementing by 1 the counts of all nodes of T that are touched while inserting σ into T.

3.1.1 Memory Restrictions while Building T

Although we can expect to have a large amount of memory and processing capacity in the runstats phase, the column of a typical database is usually very large, and the tree could grow substantially. Memory is never unlimited in practice; there are various schemes we can use to effectively implement the runstats phase.

A simple method (and the strategy we chose for our tests) is to limit the maximum size of the tree. While the tree is being built we regularly prune out nodes of small count when the number of nodes in the tree threatens to exceed the maximum M. The logic behind pruning out nodes with small count is that the strings corresponding to these nodes seldom occur in the database, and intuitively, we do not lose much information by removing these nodes. Other possibilities include least recently used-based pruning strategies [BuB] (but these do not care about the counts at the nodes), or avoiding pruning by assigning nodes to pages intelligently and paging in the nodes as done for prefetching in [CKV]. If "suffix pointers" are used, we need to be careful while pruning nodes [Kri].

[4]Tree T is a generalization of the suffix tree from Section 2 in that it is a suffix tree for a *set* of strings.

3.2 Stage 2: Suffix Tree Reduction to Get τ

At the end of Stage 1 of the runstats phase, we have a (reasonably large) suffix tree T. But for the occasional pruning we do while building the tree, we can exactly match any unit pattern in the tree to get the selectivity of the unit pattern predicate.

The main intuition for deriving the reduced tree τ is related to the intuition for pruning regularly while building the tree T: We are interested in knowing as accurately as possible how much the selectivity is when the value of selectivity is large. We get the reduced tree τ of size m from tree T by pruning out all nodes of the tree except the n nodes (that can be stored using m bytes) with the largest value of *count*. (See Figure 2.) Section 3.2.1 discusses how to determine n. We call the maximum count of any node that was pruned from tree T while building the reduced tree τ as the *prune count.*

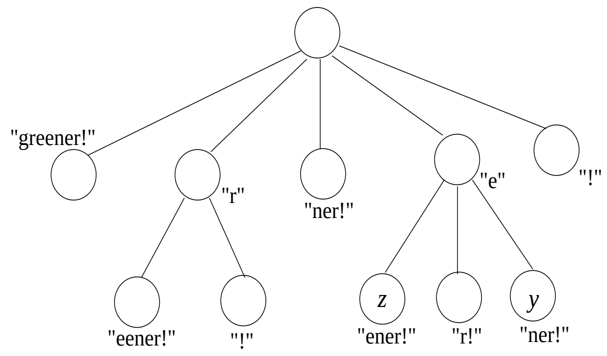
Another observation useful in reducing the time for query optimization in short online transactions is to store strings associated with a node in the metadata itself (rather than as pointers into the database).

3.2.1 Compressing Information in a Tree Node

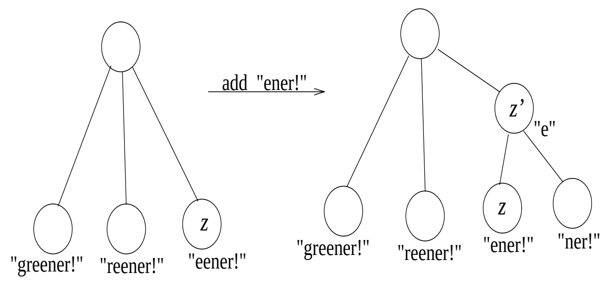
Space in the metadata and time available during query optimization phase are both limited. To pack as many nodes as possible in the metadata while keeping retrieval time small, we use some simple encoding tricks. (The number of rows in column R is denoted by $|R|$.)

We store the nodes of our reduced tree in a (contiguous) array. The children of a node are stored contiguously in the array. With each node we store the following information: the string associated with the node using a simple encoding scheme described below; the value of *count* at the node using $\log |R|$ bits; a bit to specify whether the node is a leaf, and if not, a pointer into the array to the first child of the node using $\log m$ bits; and a bit to specify whether the node is the last child in the list of children of its parent.

We store the strings associated with the node explicitly using the following intuition: Because of the higher branching we expect to see closer to the root, the strings associated with these nodes will have smaller length. We use a small number of bits b to identify the string length; e.g., with $b = 2$, we can differentiate between strings with 1, 2, 3, or > 3 characters. Strings with the most common $2^b - 1$ string lengths are stored without the trailing null character, and the other strings are stored with an explicit trailing null character. Other possi-



(a) Suffix tree for string "`greener!`"



(b) Creation process: Adding a string to a suffix tree

Figure 1: Suffix tree and its creation.

scription, see [McC].

Example 1 (See Figure 1.) A suffix tree T for string σ is a trie data structure where nodes of the trie are labeled by substrings of σ, no node (except possibly the root) has exactly one child, no two labels for the children of a node have a common prefix, and σ_i is a suffix of σ if and only if concatenating the labels on the path from the root to a leaf[2] gives σ_i. A suffix tree for string σ is obtained by *inserting* the string σ into the tree; this involves *adding* all suffixes of the string σ into the tree.

Figure 1a shows the suffix tree for string "`greener!`", where the character "`!`" is our unique end-of-string symbol. The tree is obtained by inserting string "`greener!`" into the tree; this is equivalent to adding the string "`greener!`" and all its proper suffixes "`reener!`", "`eener!`", "`ener!`", "`ner!`", "`er!`", "`r!`", and "`!`" to the tree. The substring labeling a node is called its *associated string*; e.g., "`ner!`" is the string associated with node y. The root node has the null string associated with it. The string *corresponding* to a node is the concatenation of the strings associated with the nodes on the path from the root to the node; e.g., the string corresponding to node y is "`ener!`".[3]

The strategy for creating the tree is illustrated in Figure 1b by showing how the tree changes when "`ener!`" is added to the tree after "`greener!`", "`reener!`", and "`eener!`" have been added to the tree. While adding substring σ_i (e.g., "`ener!`") to the tree, the main problem is to find the node z closest to the root such that z cannot be an ancestor of the node corresponding to σ_i. After finding z, intuitively, we "split" z to accommodate σ_i in the tree. A simple method to determine z is to start at the root of the tree and iteratively move down the tree and ahead in the string σ_i by comparing the unmatched suffix of the string σ_i against the strings associated with the children of the current node. A formal description is given in [Kri]. □

Various optimizations with respect to time and space to the basic suffix tree construction algorithm we have just described have been studied [FiG, McC]. For example, the substring σ_i associated with a node can be represented as two pointers into the original string σ. The process of determining the node z while adding a substring to the tree can be done efficiently using extra "suffix pointers" in the tree [McC].

3 The Runstats Phase

The suffix tree as described in Section 2 is a natural structure to match unit patterns in a string. For example, given the suffix tree for string "`greener!`" (Figure 1a), in order to find out whether the string "`een`" is a substring of "`greener!`", we *match* the string in the suffix tree. (The pattern to be matched is *not* terminated by the end-of-string character.) The matching process is similar to our trying to add $\alpha =$"`een`" into the suffix tree; we have a success in the matching process if we find a node z such that α is a prefix of the string corresponding to node z. We call node z as the *node of the match*. Finding out if α is a substring of σ is equivalent to the following query: Does "$*\alpha*$" match string σ, where "$*$" is the wildcard character? The suffix tree is therefore ideally suited for matching unit patterns.

Our goal in the runstats phase is to build a small structure τ to be stored in the metadata that helps us match unit patterns. We do this in two stages as described below in Sections 3.1 and 3.2.

3.1 Stage 1: Construction of Suffix Tree T

We first build a (possibly large) suffix tree T, which we then use in stage 2 to get the smaller structure τ.

[2] It is convenient to assume that the last character of the input string σ is a special end-of-string symbol (that does not appear anywhere else in the string).

[3] The difference between the string *associated* with a node and the string *corresponding* to a node is important; these terms are used throughout this paper in the sense presented here.

Algorithms to estimate selectivity can typically preprocess the database during off-hours (e.g., during the weekend), to store statistical information useful for cost estimation during query optimization as part of the *metadata* (*catalog*, in commercial relational DBMS). This pass is referred to as the *statistics generation phase* or the *runstats phase*. During the *query optimization phase*, the metadata is consulted to estimate selectivity; the processing in the query optimization phase must be minimal. Further, the space available in the metadata descriptors for any one column of the database is limited.

Models already exist in current day relational DBMS to estimate selectivity for numeric fields [ASW, HaSa, IoP, LNS, SAC, WVT]. Typically, in the runstats phase, a few numbers that "capture" the distribution of data are accumulated and stored in the metadata, as histograms, for example.

The problem of estimating alphanumeric selectivity is a natural extension to the problem of estimating numeric selectivity: the *like* predicate, and the wildcards in the patterns are natural extensions of ranges in numeric queries. To the best of our knowledge, the problem of estimating alphanumeric selectivity in the presence of wildcards has not previously been studied in a methodical way, although it is turning out to be a pressing concern.[1] This concern is corroborated by observations by practitioners in data warehousing that one half of the records have "unclean" data (through mistyping, for example) [HeS]; the unclean data also leads to large use of wildcards in queries. In practice, for alphanumeric fields, the *like* predicate along with strings with wildcards are the rule rather than the exception. The techniques used for estimating numeric selectivity are not suited for estimating alphanumeric selectivity, since the techniques for estimating numeric selectivity intimately use the fact that they are dealing with ordered measures.

1.1 Our Approach

Our strategy is based on the following interesting intuition: *The model built by a data compressor on an input text encapsulates information about common substrings in the text.* Intuitively, text data compressors (as in [ZiLa, ZiLb]) remove repetition in the input to encapsulate information better. The data structure or model built by the data compressor stores this encapsulated information. We could use a strategy similar to a data compressor's to build a structure that captures the information about substrings in a column. It is important to note that not *every* data compressor's data structure is good enough for the selectivity estimation problem, since we ultimately have to match patterns that have wildcards. We present a version of the suffix tree [McC] as an appropriate data structure for the alphanumeric selectivity problem, since it allows searches of arbitrary substrings of the strings in the database column.

[1] Programmers looking for a file whose exact name they have forgotten may empathize.

There is an emerging industry standard Transaction Processing Council (TPC) benchmark, known as the TPC-D benchmark [TPC], that involves predicates such as the *like* predicate. We study our techniques in the context of this benchmark, taking care to not take advantage of the particular structure of the benchmark. An important result of our work is that selectivity estimation for text is possible by using intuition from data compression.

In this paper, we first concentrate on trying to estimate selectivity for *unit patterns*, i.e., patterns where a string is sandwiched between two wildcards (e.g., the pattern "`*green*`" in the like predicate from (2)). Unit patterns are observed in practice to be the most common type of patterns used with the *like* predicate [TPC]. We describe strategies to deal with general patterns in Section 9.

Wang et al. [WCM] have used an interesting suffix-tree based technique to discover similar regular expression patterns in a database. Although the problem studied in [WCM] and the corresponding system constraints are quite different from our alphanumeric selectivity problem, it is interesting to note that the generalized suffix tree structure in [WCM] and our suffix-tree based data structure described in Section 3.1 are similar. This is not surprising since pattern matching is the common link between the two problems, and the suffix tree structure is natural for pattern matching.

The rest of the paper is organized as follows. In Section 2, we present the suffix tree data structure, which lies at the heart of our estimation strategy. In Section 3, we describe our strategy in the runstats phase. We present our strategies for estimating selectivity in the query optimization phase in Section 4. We present the methodology for evaluating our strategies in Section 5, and describe the results of our experiments in Sections 6 and 7. To put our results in perspective, in Section 8 we compare our method against a simple random sampling technique (where the information in the metadata is obtained by randomly choosing from the column as many strings as possible, limited by the size of the metadata). We present extensions to our strategies in Section 9. Other related issues are discussed in Section 10.

2 The Suffix Tree

A suffix tree [McC] is a trie-based data structure [Knu, Mor] used for data compression [FiG], pattern matching [Wei], and other applications. In this section we explain suffix trees with an example; for a formal de-

Estimating Alphanumeric Selectivity in the Presence of Wildcards

(extended abstract)

P. Krishnan[*]
Bell Laboratories
pk@research.att.com

Jeffrey Scott Vitter[†]
Duke University
jsv@cs.duke.edu

Bala Iyer
IBM Santa Teresa
balaiyer@vnet.ibm.com

Abstract

Success of commercial query optimizers and database management systems (object-oriented or relational) depend on accurate cost estimation of various query reorderings [BGI]. Estimating predicate selectivity, or the fraction of rows in a database that satisfy a selection predicate, is key to determining the optimal join order. Previous work has concentrated on estimating selectivity for numeric fields [ASW, HaSa, IoP, LNS, SAC, WVT]. With the popularity of textual data being stored in databases, it has become important to estimate selectivity accurately for alphanumeric fields. A particularly problematic predicate used against alphanumeric fields is the SQL *like* predicate [Dat]. Techniques used for estimating numeric selectivity are not suited for estimating alphanumeric selectivity.

In this paper, we study for the first time the problem of estimating alphanumeric selectivity in the presence of wildcards. Based on the intuition that the model built by a data compressor on an input text encapsulates information about common substrings in the text, we develop a technique based on the suffix tree data structure to estimate alphanumeric selectivity. In a statistics generation pass over the database, we construct a compact suffix tree-based structure from the columns of the database. We then look at three families of methods that utilize this structure to estimate selectivity during query plan costing, when a query with predicates on alphanumeric attributes contains wildcards in the predicate.

We evaluate our methods empirically in the context of the TPC-D benchmark. We study our methods experimentally against a variety of query patterns and identify five techniques that hold promise.

[*]Supported in part by an IBM Fellowship, by NSF grant CCR–9007851, by ARO grant DAAH04-93-G-0076, and by AFOSR grant F49620–94–1–0217. This work was done while the author was visiting Duke University from Brown University.

[†]Supported in part by NSF grant CCR–9007851 and by ARO grant DAAH04–93–G–0076.

1 Introduction

Commercial query optimizers, critical for high performance databases, are being challenged by an onslaught of SQL queries driven by advanced visualization tools typically running on personal computers. Query optimization is an integral part of databases [Dat, SAC]. For example, consider the selection predicate

$$\textit{salary between } 30\text{K and } 60\text{K} \qquad (1)$$

of a payroll database, which requests records of all employees whose salary lies between 30K and 60K. It is critical to be able to estimate *selectivity*, i.e., the fraction of rows in the database that satisfy the selection predicate. The estimation would be used by the query optimizer to determine whether to use a file scan to access all rows of the table, evaluate the predicate against each row, and return qualifying rows to the user, or to use an index on the *salary* field (if present) to access only those rows that qualify.

With the popularity of textual data being stored in database management systems (DBMS), it has become important to estimate selectivity accurately for alphanumeric fields. A particularly problematic predicate used against alphanumeric fields is the SQL *like* predicate [Dat]. For example, consider the inventory of a department store that has a *parts* table, one of whose columns, *part.color*, is the color of the part. We would like to select all parts where

$$\textit{part.color like } \text{“}\texttt{*green*}\text{”} \qquad (2)$$

where the “*” represents a wildcard that matches any sequence of symbols. In other words, we want to select all parts whose color entry has the subsequence “`green`” contained in it. The colors “`light green`”, “`dark green`”, “`greenish blue`”, would all satisfy predicate (2). We refer to selectivity of the *like* predicate as *alphanumeric selectivity in the presence of wildcards*, or simply, *alphanumeric selectivity*. The correct costing of the *like* predicate is critical because it is a prelude to many more approximate matching predicates; “sounds like” being an example, approximate matching on multiple dimensions being another.