Generalized Partial Indexes

Praveen Seshadri*

Computer Sciences Department
University of Wisconsin
Madison, WI 53706, USA
praveen@cs.wisc.edu

Arun Swami†

IBM Almaden Research Center

San Jose, CA 95120, USA
aruns@sgi.com

Abstract

This paper demonstrates the use of generalized partial indexes for efficient query processing. We propose that partial indexes be built on those portions of the database that are statistically likely to be the most useful for query processing. We identify three classes of statistical information, and two levels at which it may be available. We describe indexing strategies that use this information to significantly improve average query performance. Results from simulation experiments demonstrate that the proposed generalized partial indexing strategies perform well compared to the traditional approach to indexing.

1 Introduction

In traditional database systems, indexes are used primarily for the efficient associative retrieval of data. The database administrator (DBA) chooses to build indexes on selected columns of tuples in an RDBMS (or members of objects in an OODBMS) to speed up queries that apply predicates to those columns[1]. The choice of indexes is influenced by the number of relations in the database, the number of tuples in each relation, the number of columns in each tuple, and the number and nature of the queries against the database. Indexes also incur a cost in terms of resources (disk space and time) used to create and maintain them. Database products typically specify heuristic guidelines that the DBA should use in deciding which indexes to build[4]. This issue has also been studied as part of physical database design, and [3] describes a database design tool that helps a DBA make the index creation decisions. Conventional wisdom holds that as disk space, memory sizes and processor speeds grow, the index creation problem should become less important. Current database trends however make the problem more acute:

- The sizes of databases are growing rapidly and consequently, so are the sizes of indexes on them.
- Databases now deal with economic and scientific data which is typically enormous in size, with very wide records.
- While hardware is becoming cheaper, the cost of hardware remains non-trivial. A database system needs to get the maximum possible return on the resources invested in the system.
- The efficient processing of complex queries requires the creation of many indexes. For instance, with just 20 complex queries in the TPC-D benchmark database, the disk space needed for the desired indexes is typically greater than for the data!
- Even on databases of reasonable sizes (like the TPC-D benchmark database of 600 MB), index creation can be a matter of several hours.
- As large databases are increasingly made accessible to users, queries are becoming more ad-hoc in nature, making it difficult to build all the indexes that might be needed.

Our work is motivated by applications in which it is expensive or impossible to create every index that can possibly be useful in query processing. We ask the question: *how can the limited resources to build indexes be better utilized, so that query processing is on average as efficient as possible?* Our solution is to build *partial indexes* on relations. In traditional indexing schemes, some of the columns are not indexed. A partial indexing scheme takes this one step further, and some of the tuples in the indexed columns are not entered into the indexes. Figures 1 and 2 demonstrate this graphically. The intuitive idea is to distribute the indexing resources across the various columns, with the goal of improving the overall performance of query processing.

---

*Praveen Seshadri's research was supported by IBM Research Grant 93-F153900-000.

†Arun Swami's current affiliation is Silicon Graphics Computer Systems, Mountain View, CA 94043.

[1] Indexes may also be used for updates, for duplicate elimination or for clustered/ordered access, but we assume them to be created primarily for associative access in query processing.

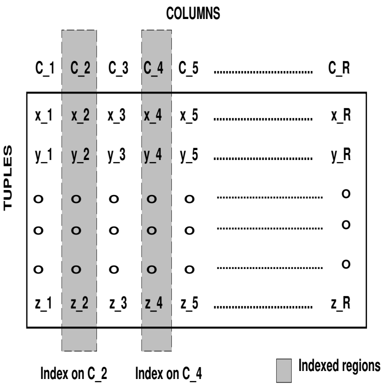

Figure 1: Traditional Indexes

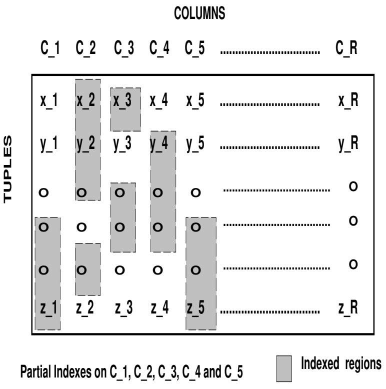

Figure 2: Partial Indexes

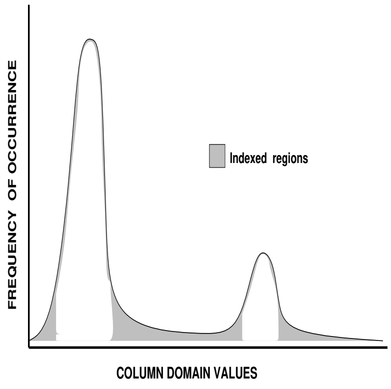

Figure 3: Using Distributions

In [9], a case is made for adding *partial indexes* to a database system. It assumes that the nature of the query predicates is known. For instance, if it is known that queries will only contain predicates of the form $X > 200$, it is sufficient to build an index on column X only for values greater than 200. It is suggested in [9] that this constraint be specified by the DBA as a clause in the command that creates the index. Since the partial indexing of [9] (which has been implemented in POSTGRES [7]) is one of the several possible partial indexing strategies described in this paper, we call our strategy *generalized partial (GP) indexing*.

A significant contribution of this paper is to demonstrate the importance of using statistics on data distributions. The data distributions in real databases are seldom uniform [2, 8], and it is often more useful to index those tuples with values in a "sparse" region of the column domain. This is demonstrated graphically in Figure 3. Further, the distribution of the query values (the actual constants in the predicates of the queries) is often quite different from the distribution of the data being queried (for instance, though most students may not have a high GPA, most queries by recruiters may be directed at the high GPA range). Finally some queries are more frequent than others and it might be more important to optimize them.

2 Classification of Indexing Statistics

Figure 4 identifies three classes of useful statistical information, and two levels at which it may be available. The figure has a column corresponding to each class of statistics. For ease of explanation, we assume that all queries involve either range or equality selection predicates.

Query Columns: A query that has the predicate $X > 200$ references the data in column X. The information on columns referenced by queries may be at one of two levels: if for every column, it is known whether or not some query uses it in a predicate, the information is "Boolean". This is the kind of information traditionally available in hard-coded applications (as represented by the lines of box 1), which build indexes on all such columns. A more sophisticated level of information may specify the percentages of queries that access each column. This corresponds to box 2 in the figure. Traditional systems can use a limited form of this information (shown by the hatched lines in box 2). A DBA could build indexes on columns in the order of descending likelihood of use in a query, until some limit on indexing resources is reached.

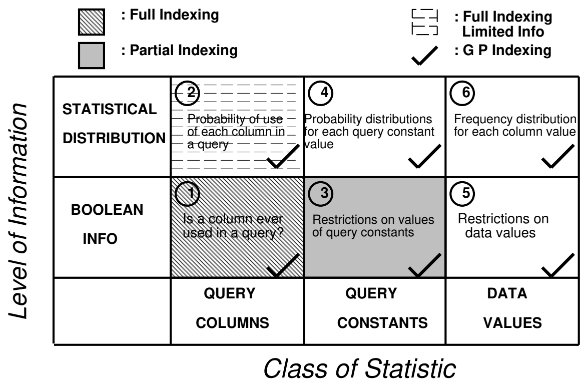

Figure 4: Classification of Indexing Statistics

Query Constants: The actual values in each query predicate $X > [value]$ may be fixed, or may follow some statistical distribution. Sometimes, limited information may be available about the values of the query constants. For instance, it may be known that though different queries may have different values for a predicate, the value is always greater than 200. This is a special case of Boolean information, and is used in the partial indexing strategy of [9] (as indicated by shaded box 3). A more sophisticated level of information could provide a distribution of query values for each query predicate on a column (box 4).

Data Values: The distribution of data values in a column is typically skewed with the more frequent values often being clustered together (in the case of numerical data). The knowledge that the columns are restricted to certain values (usually a range of values) and the actual distributions of those values (boxes 5 and 6) can be put to effective use.

Though the levels of information in each class may be different, any available information should be exploited. In this two-dimensional space of statistical information, the traditional indexing strategies primarily use the information in box 1 and limited information from box 2. The partial indexing strategy of [9] uses information in box 3 in addition. The generalized partial indexing strategies are capable of using all of this information.

3 Using Generalized Partial Indexes

An index is *partial* if it indexes a proper subset of the records in the relation. Otherwise, it is a *full* index. A generalized partial index has a collection of domain intervals for which records are inserted into the index. This is the *inclusion range* of the index. The intervals for which records are not inserted into the index constitute the *exclusion range*. We assume that some fixed number of data columns have been specified as desirable to index. The data distributions can be determined by sampling the database relations, and other statistical information can also be used if available in the system catalogs.

3.1 Generalized Partial Index Creation

Index creation is the responsibility of the DBA, and is usually performed after data loading and before queries are processed. The creation of a generalized partial index involves the following stages:

Data Sampling: The data in the relations are sampled, and a histogram of the values of each individual column is obtained, which provides a statistical estimate of the actual distribution of column values. Experiments on building equi-depth histograms using sampling [1] show that given reasonable sample sizes, the errors in such estimates are negligible.

Determining Exclusion Ranges: Assume that for each column, some fixed *window size* is specified. A window of the specified size is moved stepwise over the domain of the sampling histogram of each column. The regions in which the number of index entries in a window is greater than a specified *threshold* constitute the exclusion range for that column. This *sliding window* algorithm requires two parameters: the window size and the threshold $T\prime$, which obviously are correlated. Generalized partial indexes are created on all chosen columns by iteratively adapting the threshold value so that the available resources are fully utilized and distributed across all the indexes. Some "reasonable" choice of a window size is made which is small compared to the overall data domain. The iteration threshold T is initially set at some small value. *All* the desired partial indexes are then created; for each index, the value of the exclusion threshold $T\prime$ used is obtained by applying various *weighting factors* to T. On each successive iteration, the iteration threshold value T is increased until the overall resource usage reaches the specified resource limit.

Threshold Weighting Factors: Possible weighting factors used to determine the exclusion threshold for a column C include one or more of the following:

- If the probability P_C that a query will have a predicate on column C is known(box 2), then $T\prime = T * P_C$.
- Let the column domain be of size D and let there be N_D records in the relation. Based on Boolean query constant statistics(box 3), if query values are restricted to a range of size R_{size} containing N_R records, then let

$$T1 = T * \frac{N_R/R_{size}}{N_D/D}$$

  The actual exclusion threshold used in determining the exclusion ranges is $T\prime = \max(T, T1)$. The ratio N_D/D is the average number of tuples that have the same value for this column. If the query constants are restricted to "dense" regions where N_R/R_{size} is greater than N_D/D, the exclusion threshold used for this column needs to be appropriately increased. On the other hand, if the query constants are restricted to "sparse" regions, and the threshold T1 is significantly lower than the iteration threshold T, we make sure that the higher value is chosen by using the maximum of T and T1.
- Let $W\prime$ represent the current position of the sliding window during index creation. Based on query constant distribution statistics (box 4), if $P_{W\prime}$ is the probability that values in this window $W\prime$ will satisfy a predicate on this column, then the threshold used for this particular position of the sliding window is $T\prime = T * P_{W\prime}$.

Index Generation: Once the exclusion ranges have been identified, each record is "inserted" into each appropriate index, if and only if the column value of the record lies in the inclusion range of the index.

3.2 Query Processing

The query optimizer determines which indexes can be used to evaluate each predicate, and uses this information in generating a query plan. If the query range of a predicate (the range of domain values that satisfy the predicate) lies totally within the inclusion range, the index is classified as *usable*, otherwise it is *unusable*. By this process, a limited number of usable indexes are identified, and the optimizer can incorporate them into a query evaluation plan.

Relation Size (records)	100,000
Columns To Index	10
Page Size	4KB
Data Record Size	800 bytes
Data Records Per-Page	5
Data Distribution (all columns)	Gen.Zipf $\theta = 1.03$
Distinct Values (per-column)	10000
RID Size	8 bytes
Index Entry Size	12 bytes
Index Fill Factor	70%
Index Records Per-Page	238
Index Resource Limit	34.1 bytes * Rel.Size
Query Ranges/Domain Range	0.001

Table 1: Experimental Parameters

4 Simulation Experiments

The order of the simulation experiments mirrors the classification presented in Figure 4. Each experiment uses one particular combination of statistical information. All the generalized partial indexing strategies use the knowledge of the data distribution.

Parameters of the Simulations: We consider a single relation database. The columns of the relation are assumed to be *non-categorical*, i.e., they belong to some ordered domain. The record size of 800 bytes is modelled after applications in which each data record holds a large amount of information. The relation holds 100,000 records, resulting in an 80MB database. There are 10 columns to be indexed, each having 10,000 distinct values. The data distribution in each column is a generalized Zipfian distribution [5] (the Zipfian distribution models a significant amount of real skew data[2, 8]) with Zipf parameter $\theta = 1.03$. For the chosen database configuration parameters, this corresponds to an "80-20" distribution[5]. The indexed columns are integer valued (4 bytes) and along with the RID, each index entry has a size of 12 bytes. With an index fill factor of 70% [10], this corresponds to approximately 238 index entries per page with each index entry "occupying" 17 bytes. The resource limit for building indexes is set at (34.5 bytes * Relation Size) which is slightly more than the resources needed to build two full indexes. Each *query* is a conjunction of range predicates on the relation, each predicate being on a different column and of range 10 (i.e. 0.001 times the domain size). The last set of experiments measures the effects of varying the data skew, the query range size, the database size and the resource limits, all of which have been fixed for the rest of the experiments.

Simulator Execution: All indexed columns of each record are actually materialized in the simulator. The data in each column is distributed according to the specified distribution. Complete histograms are maintained of the distribution of data in each column. Full indexes are "created" one after another until the resource limits are met. Generalized partial indexes use the iterative approach described in Section 3 with a fine granularity window. The actual execution of a query proceeds as follows: for each range predicate in the query, if an index has been created on that column and is *usable* for the particular query predicate range, it is used to retrieve the list of RIDs that match the predicate. This is intersected with the list of RIDs obtained from the next such index, and the process is repeated until no further usable indexes remain. At this stage, the sorted list of RIDs is used to determine the actual data records that need to be retrieved from the database. The technique of index intersection has been described in [6]. Sorting the RID list before retrieving data records ensures that every database page is accessed at most once. By adopting these approaches, the simulator tries to isolate the effects of the indexing strategy used.

Simulator Cost Model: Indexes are assumed to be unclustered. It is also assumed that the queries require retrieval of the actual data records, and cannot be answered based solely on information in the indexes[6]. The indexing resource limits are expressed solely in terms of the disk space used by the indexes, and the maintenance costs are not modeled. The cost measure of the simulator is the number of unique page accesses required to process a query. For index pages, only leaf page accesses are considered. The costs of intersecting RID lists have not been taken into account and if the lists are large, these costs may be significant. CPU costs have been ignored, because in the case of simple selection queries on large amounts of data, the page access costs dominate the processing costs. Measuring page accesses is admittedly a very simple cost model. A more complex cost model could consider the advantages of sequential page access for full file scans, and the limited sequential access possible when accessing data pages from the sorted RID lists. Further, the impact of database buffering is not considered. However, the margins of difference between the various algorithms in terms of page accesses suggest that an enhanced cost model will not qualitatively alter the conclusions about the relative performance of the full and generalized partial indexing strategies.

4.1 Expmt 1: Generalized Partial vs Full

This experiment compares the full indexing scheme (box 1) with the simple generalized partial indexing scheme (boxes 1 and 6). The number of query predicates (on different columns) is varied from 1 to 9. Query constants are assumed to be uniformly distributed across the different columns. Non-uniform distributions are considered in later experiments.

Full indexing creates two indexes, while generalized partial indexing creates all 10 indexes. The graph in Figure 5 presents the average number of page accesses for the two indexing policies. It is evident that the generalized partial indexing policy uniformly performs better than full

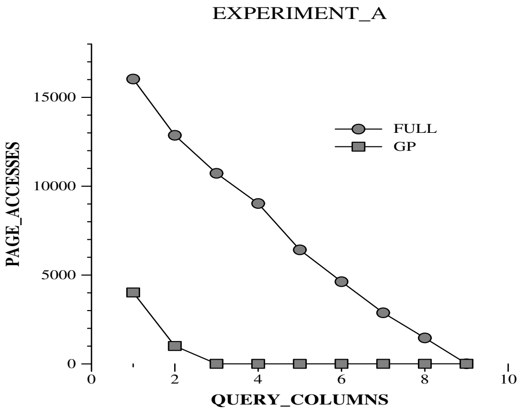

Figure 5: Experiment 1 (100000 records, Zipf=1.03)

indexing. As the number of query predicates increases, the probability that at least one of them has a usable index increases, and this reduces the average processing costs for both full and generalized partial indexing. For generalized partial indexing, since approximately 80% of the column domain is indexed and since query constants are uniformly distributed, the probability of finding a usable index becomes very high when the number of query predicates is greater than two. It should be noted that in the case when there are nine query predicates, full indexing is guaranteed to be able to find at least one usable index, although it is theoretically possible that generalized partial indexing will not. However, the probability of this happening is very small. For conciseness, we omit a graph of the standard deviation in page accesses that shows that generalized partial indexing has much lower variance in page accesses as well.

4.2 Expmt 2: Query Column Distribution

This experiment introduces additional information about the distributions of query columns (box 2). The values of the query constants are still assumed to be uniformly distributed across the column domains. The traditional indexing scheme is compared against the generalized partial indexing strategies that use this information in addition to the data distribution (boxes 1, 2 and 9). The columns are partitioned into a low frequency set and a high frequency set, each of 5 columns. Each set of columns is assigned a query column frequency. We consider the following two ratios of low to high column frequency: SMALL_SKEW= 1:2 and BIG_SKEW= 1:5. For each ratio, four different strategies are studied, while varying the number of query predicates. "Full" corresponds to the worst case scenario for full indexing, where full indexes are built on the columns of low frequency. "Full_Best" corresponds to the converse best case scenario. "GP" corresponds to the generalized partial indexing strategy that ignores the information about column frequencies. "GP_Best" is the strategy where the thresholds used in determining the exclusion ranges for each column are weighted by the query column frequencies. As in Experiment 1, the full indexing strategies create two indexes, while the generalized partial indexing strategies generate all ten (partial) indexes.

The graphs of Figure 6 and 7 show the results of the simulations. In experiments on SMALL_SKEW, there is a significant difference between the best case and the worst case for full indexing. The naive generalized partial indexing performs as well as the weighted strategy, with both of them doing much better than both the full indexing strategies. In experiments on BIG_SKEW, the relative frequencies of the different columns are more skewed. Consequently, the weighted generalized partial indexing performs slightly better than the naive generalized partial indexing when there are very few queried columns. As the number of queried columns increases, both perform extremely well. We also conducted experiments fixing the number of query predicates at 2 and varying the number of columns in the low and high frequency sets. The results indicated that the two generalized partial indexing strategies performed uniformly better than the full indexing strategies. The graphs are omitted for conciseness.

4.3 Expmt 3: Query Constant Ranges

This experiment is a variation of experiment 1 that introduces additional Boolean information about the query constants (box 3). The query constants are constrained to be within fixed ranges of the corresponding column domains. Within each such range, the query constants are assumed to be uniformly distributed. We compare traditional indexing (box 1), partial indexing of the kind suggested by Stonebraker(boxes 1 and 3) and generalized partial indexing using data distribution information as well as Boolean query constant information (boxes 1, 3 and 6). The distribution of query predicates among the columns is assumed to be uniform.

Indexing Policy	Indexes	
	A	B
Full	2	2
Partial	6	3
Generalized Partial	10	10
Generalized Partial _Best	10	10

Table 2: Experiment 3: Statistics

We present two sets of experiments, with the following characteristics: in experiment A, five of the columns are restricted to the lower 80% of the column domain (containing approximately 20% of the records). The other five columns have no restrictions. In experiment B, five of the columns are restricted to the upper 5% of the column domain (containing approximately 67% of the records). The

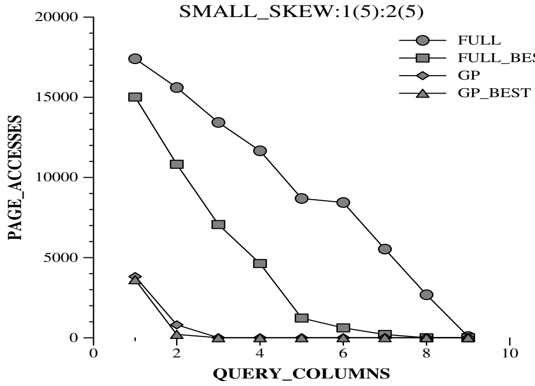

Figure 6: Experiment 2: Small Skew

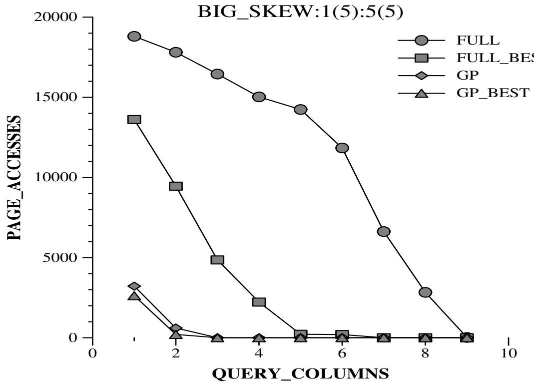

Figure 7: Experiment 2: Big Skew

other five columns have no restrictions. In each experiment set, four indexing strategies are considered. "Full" corresponds to the full indexing strategy. Note that there are no "worst case" and "best case" scenarios for full indexing because the distribution of predicates among the columns is uniform. "Partial" corresponds to the partial indexing strategy suggested by [9]. A partial index contains entries for only those tuples that lie within the query range on that column. Partial indexes are constructed over the restricted columns until the resource limit is reached. Obviously, partial indexing can create more indexes than full indexing if the query ranges are significantly smaller than the column domains. "GP" corresponds to the naive generalized partial indexing strategy that only uses data distribution statistics. "GP_Best" is the enhancement to the generalized partial indexing strategy that takes the Boolean query value statistics into account.

Table 2 shows the indexes created for each strategy for each experiment set. Note that partial indexing is able to build significantly more indexes than full indexing. In experiment A, it builds partial indexes on all the five restricted columns, and a full index on another column. In experiment B, it is able to build three partial indexes. In experiment A (Figure 8), there is little difference between the two generalized partial indexing strategies because the regions excluded by the Boolean query constant information belong to the high frequency areas of the column data distributions, and anyway belong to the exclusion ranges of the naive generalized partial strategy. Since the partial indexing strategy is able to build indexes on six columns, it performs well as the number of queried columns increases (and the probability of one of the queried columns being indexed increases). In experiment B(Figure 9), there are some very interesting results; the naive generalized partial indexing performs badly, because it does not utilize the Boolean query constant information. It builds indexes on all ten columns, totally excluding the high frequency ranges. In general (though not in this particular experiment), it can do worse even than the full indexing strategy, when the data skew is not very high! Partial indexing performs better than full indexing because it is able to build three indexes instead of two. However, the weighted generalized partial indexing strategy performs much better, because it utilizes the query constant information to weight its exclusion thresholds.

4.4 Expmt 4: Query Constant Statistics

This experiment studies the effects of utilizing knowledge of the actual distributions of the query constants. We considered two cases of query constant distributions: a uniform distribution of query constants over the domain (note that this has been the default assumption in the experiments thus far), and a distribution of query constants that mirrors the data distribution (i.e., a value that is more likely to occur in the data is more likely to occur in the queries as well). The results indicated that in the first case, the weighted generalized partial indexing strategy reduced to the naive strategy, while in the second case, it reduced to the full indexing strategy. Thus, the weighting factor was able to incorporate the query constant distribution information. We have omitted a detailed discussion due to space constraints.

4.5 Expmt 5: Resource Usage

The goal of this experiment is to determine the relative amount of resources used by generalized partial indexing to provide performance similar to full indexing. The experimental parameters used for full indexing are those of Experiment 1. For each particular x-axis point (number of query columns), the resource limit was manually varied so that the performance of generalized partial indexing was almost identical to that of full indexing with the original resource limits that allow two full indexes to be created. The experiment was repeated for two different values (0.86 and 1.03) of the Zipf parameter. The graph in Figure 10

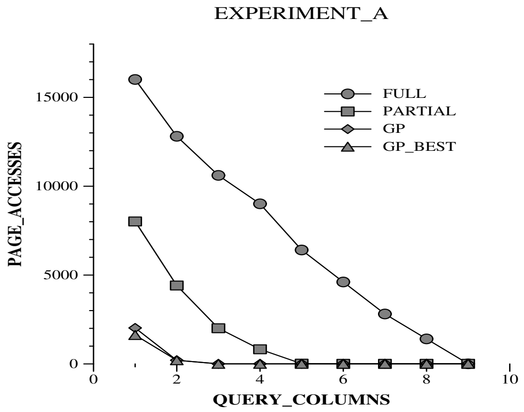

Figure 8: Experiment 3.A: Lower 80%

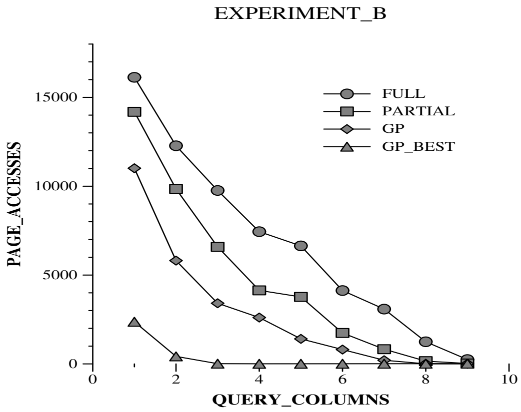

Figure 9: Experiment 3.B: Upper 5%

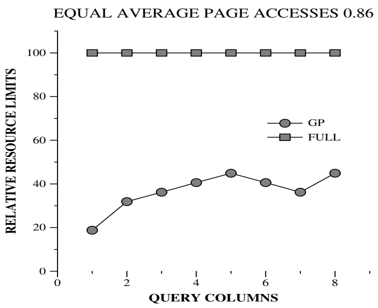

Figure 10: Expmt 5: Zipf 0.86

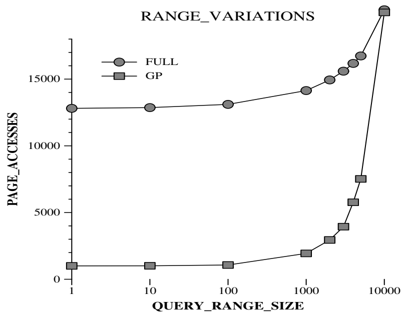

Figure 11: Range Size Sensitivity

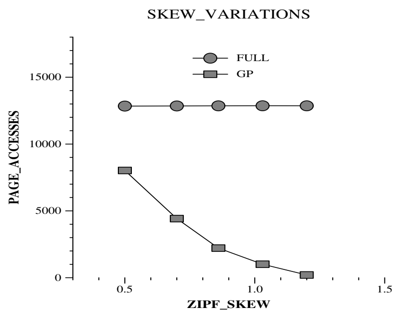

Figure 12: Skew Sensitivity

displays the ratio of the resources needed for generalized partial indexing to the resources needed for full indexing, for Zipf 0.86 (the more skewed case of 1.03 has been omitted since it merely displays a more pronounced difference in resource requirements).

While the actual ratio varies, it is always between 18% and 44%. When the number of queried columns is low, every additional query column helps the performance of full indexing a little more than it helps generalized partial indexing, causing the graph to slope upwards. The increase in the number of query columns increases the probability that the indexed columns will be accessed, and this increase is more marked in full indexing, because generalized partial indexing does a better job of spreading out its indexing resources. When the number of queried columns crosses five, the likelihood of generalized partial indexing finding a usable index to process the query becomes very high, and this factor makes it perform much better than full indexing. The graph therefore slopes downwards again. Finally, when the number of queried columns is eight, the same effect begins to affect full indexing which has full indexes on two columns. Therefore, the performance of full indexing improves sharply, causing the ratio to increase again.

4.6 Sensitivity Experiments

- With the parameters of Experiment 1, and with two query columns, we varied the size of the query ranges from one till the size of the database. The graph in Figure 11 shows the corresponding performance of full and generalized partial indexing. For smaller range sizes, generalized partial indexing is clearly better. As a special case, *generalized partial indexes perform well on equality predicates*. For large range sizes, the advantages of performing a sequential file scan will probably make that a more desirable alternative than indexed access anyway.
- With the parameters of Experiment 1 and two query columns, we varied the value of the θ parameter of the generalized Zipf distribution used to represent the data. Figure 12 shows the results of these experiments. When the data distribution becomes more skewed, generalized partial indexing performs increasingly better than full indexing.
- The relative performance is not sensitive to the exact setting of the resource limits, as long as the desired resources for building indexes are much larger than the available resources, i.e., as long as the number of full indexes that can be built is small compared to the number desired.

- The relative performance is unaffected by variation in the relation size, as long as the resource limits are proportionately varied.

5 Conclusions

The simulation studies have shown that generalized partial indexes deserve further investigation as a means of improving query processing efficiency. There are, of course, a number of other issues to address; some of these are interesting areas of future research. Partial indexes do not support clustering, uniqueness, or ordered access to data, in the manner of a full index. Guidelines must therefore be developed to decide when partial indexes are desirable. The discussion of partial indexes in this paper does not deal with issues like updates, multi-attribute indexes and compression techniques. Instead of a sliding window technique that is influenced by the choice of window size, other heuristics could possibly be used to determine the exclusion ranges of partial indexes. Finally, this paper has not dealt with join processing algorithms that use generalized partial indexes; this is a topic that we are currently investigating.

Acknowledgements

We thank Mike Carey for extensive comments and suggestions that greatly helped improve this paper. Bill Cody and Honesty Young also provided useful feedback on an earlier draft.

References

[1] D. J. DeWitt and J. F. Naughton and D. F. Schnieder. Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting. Technical report, Computer Sciences Department, University of Wisconsin-Madison, August 1991. CS Technical Report 1043.

[2] C. Faloutsos and H. V. Jagadish. On B-tree Indices for Skewed Distributions. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 363–374, Vancouver, British Columbia, 1992. Morgan Kaufman.

[3] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical Database Design For Relational Databases. *ACM Transactions on Database Systems*, 13(1):91–128, March 1988.

[4] IBM Corporation, White Plains, NY. *DB2 System Planning Administration Guide*, 1993. IBM Publication No. SC26-4085.

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

[6] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques. In *Proceedings of the 1990 International Conference on Extending Database Technology*, pages 29–43, March 1990.

[7] N. E. Olson. Partial Indexing in POSTGRES. Technical report, Department of Electrical Engineering and Computer Science, U.C.Berkeley, June 1993. Master's Report.

[8] G. C. Steindel and H. G. Madison. A Benchmark Comparison of DB2 and the DBC/1012. In *CMG '87, International Conference on Management and Performance Evaluation of Computer Systems*, pages 360–369, Orlando, FL, 1987. The Computer Measurement Group, Inc.

[9] M. Stonebraker. The Case for Partial Indexes. *ACM-SIGMOD Record*, 18(4):4–11, December 1989.

[10] A. Yao. On Random 2-3 Trees. *Acta Informatica*, 9(2):159–170, 1978.