THE CASE FOR PARTIAL INDEXES

Michael Stonebraker

Department of Electrical Engineering
and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Current data managers support secondary and/or primary indexes on columns of relations. In this paper we suggest the advantages that result from indexes which contain only some of the possible values in a column of a relation.

1. INTRODUCTION

Most commercial data managers support indexes on columns of data base objects. For relational systems, such indexes are on columns of relations and may be primary (or clustered) as well as secondary (unclustered). In addition, both hashed and B-tree indexes are available on various commercial offerings. In older systems, such as IMS, indexes on fields in segments are provided as well as indexes which contain an entry for each segment giving the value of a field in a parent or dependent segment [DATE83]. Such indexes, which contain information relevant to more than one type of object are somewhat analogous to the join indexes proposed by [VALD87] and to the links in System R [ASTR76].

However, it appears that few people have suggested indexes on less than a complete column. In this paper we indicate the characteristics of such indexes and the uses to which they can be put. Section 2 discusses the composition of partial indexes, and then Section 3 turns to uses for them in normal data processing. We discuss the idea in the context of a relational data base system, though it appears to be equally useful in a DBMS for any data model.

2. PARTIAL INDEXES

2.1. Specification

We propose that a data manager accept an indexing command of the form:

create index-type INDEX on relname
(column-name) where qualification

This stylized syntax omits details (such as the fill factor and whether one or more compression techniques is to be used). For example:

create B-tree INDEX on EMP (salary)
 where salary < 500
create hash INDEX on EMP (salary)
 where age = 50
create B-tree INDEX on EMP (salary)
 where age = 50 and salary < 500

The first example suggests a salary index only for those employees with salaries under $500. The second example would build a hashed salary index for employees with ages equal to 50 while the third would build a B-tree index for employees satisfying both conditions.

We will assume that the qualification does not contain join terms or aggregates. Moreover, it is assumed to be a conjunct of simple terms, i.e:

i-clause-1 and i-clause-2 and ...
and i-clause-m

where each clause is of the form

field operator constant

Hence, the qualification can be evaluated for each tuple in the indicated relation without consulting additional tuples.

Each index is assumed to use a collection of fields, F1,...., Fn as the **organizing** keys. Hence, the DBMS is expected to build the index structure using these keys. Each field Fi may or may not be mentioned in the qualification associated with the index. The second example above is an index where the organizing field is not mentioned in the qualification.

We will also assume that B-trees and hashing are the access methods that are of interest. However the ideas all generalize to other access methods using techniques along the lines of [STON86a].

2.2. Query Optimization and Execution

A query optimizer can use such indexes with only modest extensions. We first assume that the user query is processed into disjunctive normal form and that each disjunct is treated as a separate query. Hence, each query to be processed has a qualification of the form

clause-1 and clause-2 and ...
and clause-n

We will also ignore the presence of aggregates in user commands.

For each scan of a relation which the optimizer investigates performing, it must first ascertain which indexes can be used. Two tests must be run, one on index usability and one on predicate inclusion. The usability test is the same as the one required in [SELI79]. Specifically, a hash index is usable if every organizing key is mentioned in the predicate of the user's query in a clause of the form:

field = constant

A B-tree index is usable if there exists a K such that all organizing keys to the left of the Kth key have the property that they appear in a clause in the user query of the form:

field comparison-operator constant

The predicate inclusion [illegible] must ascertain for those usable indexes which ones have a predicate that includes the predicate in the user query. In particular, an index with qualification, IQUAL includes a user query with a qualification, QUAL if:

IQUAL contains QUAL

For example, if a user performs the query:

retrieve (EMP.name)
where EMP.salary < 400

then the first index includes this query while the other two do not.

A fast test for this inclusion property is the following. Consider all the fields mentioned in all clauses of the user query. An index is non inclusive for the query if its qualification contains any field not mentioned in the user query. For the remainder of the usable indexes, we must perform a further check to ascertain which ones have the inclusion property. Specifically, an index includes a user query if for each clause in the index, i-clause-j, there exists a clause in the user query, clause-u, such that

i-clause-j contains clause-u

This test can be run in a straight forward way for the comparison operators which accelerate B-tree and hashing access.

If several indexes are usable and have the inclusion property, the optimizer must now choose the most profitable one to use. It can use straightforward generalizations of the computations in [SELI79]. Consider a hash partial index on organizing keys, F1, ..., Fn. Because the index is usable, there exist clauses in the user query, Q(F1), ..., Q(Fn), each with the property noted above. Also, some of the organizing keys may be included in a clause in the predicate for the partial index. Let these clauses be IQ-1, ..., IQ-j and let:

user-clause-selectivity = selectivity
[Q(F1)] * ... * selectivity[Q(Fn)]

partial-selectivity = selectivity
[IQ-1] * ... * selectivity[IQ-j]

The estimated number of data records examined by using this index is therefore:

E = index-cardinality * user-clause-selectivity / partial-selectivity

E can be used in the normal way by a conventional optimizer when making access path decisions. For B-tree indexes the computation is similar and is omitted for brevity.

For example, suppose there are 10,000 records in the index for salaries under 500, the selectivity of the clause

salary < 500

is 0.2 and the selectivity of the clause

salary < 400

is 0.15. In this case

E = (10,000) * (0.15) / (0.2) = 7,500,

i.e. 7500 index records and corresponding data records will be examined in solving the query through the partial index.

Some DBMSs can make use of multiple indexes when processing a single query. For example, if a user issues the query:

retrieve (EMP.name) where
EMP.age > 40 and EMP.salary < 2000

then it is possible to access an age index to obtain a list of record identifiers for older employees followed by an access to a salary index to obtain low paid employees. These two lists can be subsequently intersected to find the actually qualifying employees who are then fetched from disk. When partial indexes are present this strategy must be more carefully considered. Consider two partial indexes:

create B-tree index on EMP (age)
where salary < 2100
create B-tree index on EMP (salary)
where salary < 2100

Both indexes are usable and include the above user query. However, the conditional probability of a record appearing in the second index given that it appeared in the first index is 1.0. Consequently, the second index gives no additional discrimination and its use should be avoided.

Regular indexes are often heavily correlated. For example, an age index and salary index are positively correlated because older employees tend to earn larger salaries. However, partial indexes are even more likely to be correlated than normal indexes. Consequently, the typical assumption made by query optimizers that indexes are independent is untenable in a partial index environment. An estimate of index correlation can be readily obtained because each index is only partial. Namely, have the statistics module obtain a random sample of N tuples from the relation being indexed. For each tuple, evaluate the two index qualifications and record whether the tuple satisfies neither, one, the other, or both qualifications. An estimate of the correlation C(I1, I2) of the indexes I1 and I2 is then:

tuples satisfying both indexes /
(tuples satisfying I1 * tuples satisfying I2)

After obtaining an estimate for the correlation, the expected number of records examined in the data relation for a given query if both indexes are used is:

$$E(I1) * E(I2) * C(I1,I2)$$

where E(I1) and E(I2) are the number of data records fetched if either index was used in isolation.

With regard to updates, it is evident that the run-time system must update a partial secondary index only if it inserts or deletes a tuple which satisfies IQUAL. In addition, crash recovery must handle partial indexes using the same techniques that are used for current conventional indexes. If index records are physically logged as in DB2, then this practice can be trivially extended to partial indexes. If index modifications are not logged, then the code which runs at recovery time must ascertain which indexes are affected by the change encoded in any log record and undo or redo the appropriate index operations.

3. USES FOR PARTIAL INDEXES

Partial indexes such as described above are clearly useful in normal query processing as will be discussed in Section 3.1. However, they have several other uses that are less self-evident. Consequently, we discuss the use of partial indexes in incremental indexing techniques in Section 3.2 through 3.4. Then in Sections 3.5 through 3.8 we discuss other uses of partial indexes not related to incremental indexing.

3.1. User Indexes

Often a user application has the characteristic that portions of a range of key values are uninteresting. In this case one wants to index only the part of the range that users ask queries about. For example, if a large fraction of the tuples have default or null values for a particular key, one might want to access only the non-default values. Specifically, if 80 percent of the employees have not been given a salary and have a value of zero, then it would be prudent to index only the positive salaries. In this way, no system resources are spent indexing the uninteresting values.

Another example concerns purchase orders (POs). Suppose a financial officer is only interested in POs over $100,000, since small POs rarely make a large financial impact. Partial indexes allow only the POs that will actually be queried to appear in the index. Again no resources are spent indexing data that will not be examined later.

A final example is the use of partial indexes for exceptions. Consider a sex field with normal values, male and female. Since the discrimination of the index is too low, a query optimizer would never choose to use a full index on the sex field. However, there may be a few employees who have a sex of unknown and it might be useful to have an index solely on the exceptional values, i.e:

```
create hash INDEX on EMP(sex)
 where sex = "unknown"
```

In this way, a data administrator trying to clean up the EMP data would have an index into the employees who required values.

The following simple model quantifies what subranges of a value range are desirable to index. Assume that the value range is an interval [A,B] using the operator < to order the values. Furthermore assume that equality is defined and that queries that involve the field F are of the form:

```
retrieve (Target-List) where F = constant
```

Assume the cumulative distribution function for the constant in the qualification is G1(X), i.e.:

$$G1(X) = \text{probability (constant} < X\text{)}$$

Furthermore assume inserts (or updates) specify a new value for the field F and that the distribution function for new values is:

$$G2(X) = \text{probability (newvalue} < X\text{)}$$

Assume the cost of a sequential scan is S, the cost of an indexed access is A, and the cost of an indexed update is I. Lastly, for each retrieval assume there are an average of U updates.

In this case an interval [a,b] should be included in a partial index only if:

$$[G1(b) - G1(a)] / [G2(b) - G2(a)] > U * I / (S - A)$$

Besides restricting the index to contain only values of interest to the user community, a partial index has an added benefit. Suppose the data manager only uses the maximum value, the minimum value and the tuple count to obtain needed selectivity estimates. If the data is uniformly distributed, then such estimates are accurate for full indexes. On the other hand, if skew is present in the data, then selectivity estimates can been exceedingly inaccurate. Although the importance of accurate selectivities can be questioned for complex queries [KUMA87], they are clearly essential in processing complex boolean qualifications on a single relation [LYNC88]. Hence, a partial index which leaves out a region of high skew, will result in more accurate selectivity estimates for queries that can use the partial index.

3.2. Incremental Index Building

In many DBMSs a batch run must be used to build a secondary index. Generally this will require that the relation being indexed become unavailable for the duration of the build process, often hours of elapsed time. Such environments do not make data highly available. Obviously, one can build indexes without locking the relation by using a variation on techniques for fuzzy dumps. The basic idea would be to process the relation in a single sequential scan, making known in a shared global variable (MARKER) the tuple identifier currently being examined. A concurrent user who makes an update in advance of the MARKER need make no modification to the index being built. However, if the update is behind the MARKER, then the run-time system must update the index being constructed if the key is changed. In addition, suitable footprints must be left in the log to allow both the index update and the data update to be undone in case of a transaction abort or other failure. We now present an easier technique using partial indexes.

Consider the data relation to be a collection of pages, P1, ..., Pn. Divide this collection into intervals of size I > 0 pages. Lock the first interval, and then build the index for the data tuples in this interval. At the end of this process mark the index as having the qualification:

tuple-identifier < X

where X is the first tuple identifier on the next page to be examined. This index can remain in place for any period of time and updates will be automatically propagated to the index as necessary. At some later time the next interval can be processed, and over time a complete index built. Like techniques based on fuzzy dumps, this approach has the advantage relative to building indexes via a batch run that the relation is not made unavailable during the index build and commitment of system resources to build the index can be deferred to period of light system load.

3.3. Incremental Modification

Often a user wishes to convert an index from one access method to another, say from hashing to B-tree or vica-versa. In addition, one often wants to rebuild an existing B-tree index which has become physically declustered after a large number of page splits and regroups. Partial indexes are well suited to this task.

To convert from a B-tree index on a key to either a rebuilt B-tree index or a hash index on the same key, one can proceed as follows. Divide the key range of the index into N intervals of either fixed or varying size. Begin with the first interval. Lock the interval and construct a new index entry for each tuple in the interval. When the process is complete, unlock the interval. The new index is now valid for the interval

key < VALUE-1

where VALUE-1 is the low key on the next index page to be examined. The old index can be considered valid for the whole key range or it can be restricted to:

key >= VALUE-1

In this latter case, the space occupied by the index records of the first interval can be reclaimed. If the intervals are chosen to be the key ranges present in the root level of the old B-tree, then this space reclamation can occur without destroying the B-tree property for the old index.

The query optimizer need only be extended to realize that the two indexes together cover the key range. Hence, if a query must be processed with a qualification of the form:

where VALUE-3 < key < VALUE-4

it may be necessary to begin the scan in the new index and then continue it in the old index when new index records are exhausted. There is little complexity to this optimizer extension.

At one's leisure, the remaining N-1 intervals can be processed to generate the complete index.

To convert from a hash index to another hashed index or a B-tree index, the procedure is similar. One simply divides the hash buckets into N groups and processes the groups one-by-one as above. After the first interval has been finished, the new index has a qualification:

hash (key) in range-of-buckets

The old index can be changed to:

hash (key) not-in range-of-buckets

During the time that both indexes are in existence, each can be used for a portion of the queries with qualifications of the form:

where key = value

Unfortunately, if the new index is a B-tree, then queries which involve ranges of values on the key in question cannot be solved using the new index until it is completely constructed. Of course, this procedure requires that the qualifications discussed in Section 2 be slightly generalized to allow a function of a key to appear.

3.4. Building Indexes as a Side-effect of Query Processing

Suppose one wishes to have a partial index built on a field F of a relation R with index qualification IQUAL. Further suppose as a result of preceding queries, part of the index has been constructed, namely those records satisfying PART-QUAL. The remainder of the index, i.e. those records such that:

IQUAL and not PART-QUAL

remain to be constructed. Further suppose the DBMS processes a user query which involves the relation R. Specifically, suppose that the sub-query which entails R is of the form:

retrieve TL(R) where USER-QUAL

If so, then the execution of this query presents an opportunity for partially or completely satisfying the indexing goal. If

USER-QUAL contains (IQUAL and
not PART-QUAL)

then the index can be fully built as a side effect of query processing. Whenever one finds a tuple which satisfies USER-QUAL, one must insert an additional check for

IQUAL and not PART-QUAL

If the check is successful, then an insertion should be made in the index. The only change which must be made to the query plan is that the fields present in the qualification, IQUAL and not PART-QUAL, must not be projected out during earlier processing so the needed check can be performed. At the conclusion of the user query, the index has been fully extended to the qualification, IQUAL.

On the other hand, if

USER-QUAL and (IQUAL and not
PART-QUAL) is non-empty

then the check must be included as above. At the conclusion of the user query, the index will have the qualification. Q:

Q = PART-QUAL or [IQUAL and
not USER-QUAL]

Over time the desired index can be built up as a side effect of query normal processing. This technique has the advantage that it does not require locking any records that are not already locked by the user query. Hence, the index can be constructed with no extra unavailability of the relation. Moreover, building the index is accomplished without reading any records that were not already required by some user query. Lastly, the writes required to build the index are spread over several user commands, so the disk load of index creation is dispersed over several commands.

The disadvantage of the above technique is that after a few applications, the partial qualification, Q, becomes an exceeding complex boolean expression. Excessive CPU time may be consumed in the required check built into the technique. To avoid this problem, the following special case is probably a better practical idea.

Suppose one wants to build a hash or B-tree index with qualification IQUAL on a particular key, K2, and one has available a second B-tree index on a key, K1. Suppose further that the desired index qualification, IQUAL, for the index being constructed is of the form:

LOWER < K2 and K2 < UPPER

Now suppose a user query is processed that contains a qualification of the form

where K1 > VALUE-1 and
K1 < VALUE-2

and suppose the query optimizer has decided to solve the query using the index on K1. For each qualifying tuple, an insert can be made in the K2 index being constructed for those records with suitable values. At the end of the operation, the qualification for the K2 index is:

LOWER < K2 and K2 < UPPER and
VALUE-1 < K1 and K1 < VALUE-2

Over time, VALUE-1 and VALUE-2 can be extended to cover the whole K1 range and the partial index fully constructed.

During the intervening steps the partial qualification, Q, remains much simpler than in the general case and therefore has the possibility of being useful during normal query processing.

3.5. Corrupted Indexes

If a page or a collection of pages in a secondary index is corrupted through a software or hardware failure, the system can isolate the key range or collection of hash buckets which corresponds to the corrupted area. Then the system can refine the index definition to exclude the keys range(s) or hash buckets involved. At one's leisure, the index can be re-extended to the complete range. Again a partial index is appropriate for this task.

3.6. Result of User Queries

In certain application areas, such as information retrieval (IR), a user often asks a query and then later on decides to add qualification to a previous query to refine the search. In IR this usually happens when the first query produces a response that contains too many citations. Current IR systems often keep a list of pointers to qualifying records of previous queries, and in this way the search can be refined without redoing all the original work.

Since IR systems are typically read-only, there is no need to worry about the consistency of this list of pointers. However, in a situation where updates are present, this list-of-pointers technique will become inaccurate. For example, an accounts receivable officer might want to find all unpaid purchase orders over $100,000. Finding a large set, he might want to refine the search using additional criteria. During this session additional qualifying purchase orders might be put into the data base. The list of pointers would not get extended with references to these new tuples.

If the list of pointers is constructed as a partial index, then this problem is avoided. Moreover, the partial index could conceivably be used to help solve the queries of other concurrent users. Although this is unlikely in IR applications, it is possible to think of situations where this would be a helpful tactic.

3.7. Precomputation of Procedures

Some systems advocate storing procedures as objects in the data base, e.g. POSTGRES [STON86b], Sybase, and the Britton-Lee IDM [EPST80]. In addition, POSTGRES is exploring precomputing the value of a procedure before the user requests that it be evaluated. Consider, for example the procedure:

```
retrieve (EMP.name) where
EMP.salary > 1000 and
EMP.age < 40
```

Precomputing this procedure produces a list of names which are cached by the run-time system. Unfortunately, if someone receives a salary adjustment or his age changes, the cached answer may become invalid. Either the cached procedure must be invalidated and then reconstructed, which will be VERY time-consuming and wasteful of resources, or a way must be found to update the cached procedure value. The algorithm to update the cache is the same one used to update a materialized view [BLAK86, HANS87] and is not particularly easy to implement.

Alternately, the above query corresponds to a partial index on employee names with a qualification:

```
salary > 1000 and age < 40
```

Hence, the algorithm sketched earlier can be used to easily update the index. Consequently, precomputed procedures can sometimes be represented by partial indexes.

Of course. if the query to be precomputed involves joins or aggregates, then this technique no longer works. However, an important class of computations can be efficiently supported.

3.8. Substitute for List Processing Techniques

It is often argued that queries such as

```
retrieve (EMP.name) where
EMP.salary > 1000
and EMP.age < 40
```

should be processed by consulting a salary index for a list of records which satisfy the first clause followed by investigation of an age index for lists of records which satisfy the second clause. The next step is to intersect the two lists of record identifiers to produce the actual qualifying ones. The last step is to access the data records which qualify to obtain employee names.

The presence of partial indexes allows a possibly competitive alternate strategy. In particular, suppose one constructs a partial index on salaries as follows:

```
create B-tree INDEX on EMP(salary)
where EMP.age < VALUE
```

If VALUE > 40, then the index is usable in solving the above query. One need only inspect salary records in the partial index for ones over 1000. Then, the data records will be examined to find employees who actually satisfy the qualification. Notice that there will be a collection of "false drops", i.e. employees with ages between 40 and VALUE. These records will be accessed even though they don't qualify and represent wasted effort. On the other hand, sorting two lists of record identifiers and then merging them has been removed. The performance of partial indexes depend on whether one of the indexes is clustered. Hence, the analysis will now be done for the cases:

partial index is clustered
partial index is not clustered

As a simplifying assumption, we will only count I/Os and will make the traditional assumption that only limited main memory is available for buffering.

Suppose there are:

N	records in a relation R
B	block size of disk blocks
K	key width of indexed fields
E	number of records accessed through the partial index
P1	selectivity of the first clause in the user query
P2	selectivity of the second clause in the user query
P3	E / N

Define:

$$X1 = (N * P1 * K) / B$$
$$X2 = (N * P2 * K) / B$$

When using multiple single indexes, the DBMS must retrieve and perform a disk sort of the relevant portions of the two indexes which requires:

$$\# \text{ I/Os } = X1 * \log X1 + X2 * \log X2$$

Next the two lists must be merged and then the relevant records fetched. Ignoring the cost of the data fetch (which must be done anyway), the total cost for the list intersection approach is:

$$\text{List \#I/Os } = X1 (1 + \log X1) + X2 (1 + \log X2)$$

This cost is independent of whether or not the indexes are clustered.

When the partial index is not clustered, the cost is calculated as follows. First the relevant portion of the partial index must be fetched at a cost:

$$P3 * X1 / P1$$

Then, the potentially qualifying records must be fetched. Since the number of actual qualifying records must be fetched anyway, we count only the wasted effort to fetch the false drops which, assuming that they are not clustered, is:

$$N * (P3 - P1 * P2)$$

Hence, the total cost of the partial index approach is:

$$\text{Partial \#I/Os } = P3 * X1 / P1 + N * (P3 - P1 * P2)$$

On the other hand, if the partial index is a clustered index, then the false drops will be intermixed with qualifying data records and will not require any extra disk I/O. Hence, only the cost corresponding to the first term in the above equation must be paid.

If P1 = P2 = 0.1, P3 = 0.02, N = 1,000,000, B = 4096 bytes, and K = 20 bytes, then log X1 = log X2 ~ 4 for reasonable bases for the logarithm. Hence,

Partial # I/Os (clustered)	100
Partial # I/Os (unclustered)	10,100
List # I/Os	~5000

Hence, the partial index approach is quite competitive with the list intersection technique. It outperforms the list intersection technique if the partial index is clustered. Moreover, an unclustered index will have similar performance to list intersection as long as P3 is not far from P1 * P2. Of course, it deteriorates to the conventional approach as P3 approaches P1.

4. CONCLUSIONS

This paper has suggested including partial indexes in data base management systems. They entail only marginal extra complexity and can help solve a collection of problems that any DBMS faces. These include indexing only intervals of interest to users, incremental build and modification of indexes, indexes on results of user queries and alternatives to list processing code applied to multiple indexes.

REFERENCES

[ASTR76] Astrahan, M. et. al., "System R: A Relational Approach to Data," ACM-TODS, June 1976.

[BLAK86] Blakeley, J. et.al., "Efficiently Updating Materialized Views," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[DATE83] Date, C., "An Introduction to Database Systems," (3rd edition), Addison-Wesley, Reading, Mass., 1983.

[EPST80] Epstein, R., and Hawthorn, P., "Design Decisions for the Intelligent Data Base Machine," Proc. 1980 National Computer Conference, Anaheim, CA, June 1980.

[HANS87] Hansen, E., "A Performance Analysis of View Materialization Strategies," Proc. 1987 ACM-SIGMOD Conference on Management of Data, San Francisco, CA, May 1987.

[KUMA87] Kumar, A. and Stonebraker, M., "The Effect of Join Selectivities on Optimal Nesting Order," SIGMOD, Record, March 1987.

[LYNC88] Lynch, C., "Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values," Proc. 1988 VLDB Conference, Los Angeles, CA, Oct. 1988.

[SELI79] Selinger, P. et. al., "Access Path Selection in a Relational Data Base System," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[STON86a] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. Second International Conference on Data Base Engineering, Los Angeles, CA, Feb. 1986.

[STON86b] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[VALD87] Valduriez, P., "Join Indices," ACM-TODS, June 1987.