Database Tuning with Partial Indexes

Alain D. Fuentes[1], **Ana Carolina Almeida**[2], **Rogério Luís de Carvalho Costa**[1], **Vanessa Braganholo**[3], **Sérgio Lifschitz**[1]

[1]Departamento de Informática – PUC-Rio, Rio de Janeiro, Brazil

[2]Departamento de Ciência da Computação – UERJ, Rio de Janeiro, Brazil

[3]Instituto de Computação - UFF, Niterói, Brazil

{afuentes,rogcosta,sergio}@inf.puc-rio.br
ana.almeida@ime.uerj.br, vanessa@ic.uff.br

Abstract. *Database tuning usually involves indexes, materialized views, partitioning, query rewriting and other techniques. One strategy that presents good results for performance improvements is the use of partial indexes. However, partial indexes have not been used for database tuning in the past. This is because the search space for partial indexes is exponential in the number of attributes and tuples of the table. In this paper, we address this problem by proposing an optimized strategy to select partial indexes. The optimization relies on reducing the amount of logic reads. We explain how to select the indexable attributes and their corresponding restrictions through a formal procedure. We implement our strategy to illustrate the benefits of partial indexes for tuning issues. Results are promising.*

1. Introduction

Database applications have become increasingly complex and varied. These involve very large datasets and a high demand for good performance. The problem has always been on how to decrease query response time while increasing the throughput (number of queries executed per unit of time). In this context, tuning the physical design of database systems has been proved to be extremely important in improving the performance of database systems [Shasha and Bonnet 2002].

Index tuning, as part of the physical database design, is the task of selecting, creating, deleting, and rebuilding index structures to reduce workload processing time. Among the activities related to database tuning, the adjustment of index structures represents one of the most relevant. This fact stems from the great benefit that these structures bring to the performance of database systems, since it can substantially reduce the execution time of the queries, including updates [Shasha and Bonnet 2002].

Current index tunning approaches use regular (or complete) indexes in detriment of partial index. A partial index [Stonebraker 1989], which is present in some of the major Database Management Systems, indexes a subset of the tuples of a table instead of indexing the complete set of tuples. However, the search space for defining a partial index is exponential in the number of attributes and tuples of the table, which explains why it has not been used by database tuning approaches up to now. In this paper, we propose an optimized strategy to select partial indexes. It is based in use cases oriented to reduce the amount of logic reads. Our strategy contemplates the use of both partial and complete indexes in automatic database tuning.

It should be noted that the search for partial indexes as a tuning action includes not only all steps followed when searching for complete indexes but also the definition of the subset of tuples that will be accessed. Nevertheless, we propose a multi-column partial index approach and show that there are situations where it is worth to consider partial indexes rather than complete indexes for performance issues.

This paper is organized as follows. Section 2 defines partial indexes, while Section 3 discusses related work. In Section 4 we present a strategy to select partial indexes and combine them with complete indexes in the tuning process. Some experimental results are given in Section 5 and Section 6 concludes, listing our main contributions.

2. Preliminaries

Stonebraker defines partial indexes as constrained indexes with a WHERE clause, which defines a subset of tuples to be indexed on a table [Stonebraker 1989]. For example, we could define a partial index PI for table T with two columns A and B, as follows. In this case, only tuples from T that have attribute B valued $'X'$ or $'W'$ would be indexed.

```
CREATE INDEX PI ON T (A, B) WHERE B = 'X' or B = 'W';
```

A partial index P can be used in the execution of a query Q if and only if the predicate of Q logically implies the conditional expressions of the partial index P. For instance, consider query Q below that is run over table T:

```
SELECT * FROM T WHERE B = 'X'
```

It is easy to note that the partial index PI above contains enough information to find the tuples of T satisfying query Q, since the selection predicate of Q logically implies the selection predicate of PI. We denote this set of tuples $T(Q)$.

Partial indexes are beneficial because they can avoid indexing frequent values. When a query retrieves a set of values of more than a small percentage of all table rows, the DBMS does not use the index. Then there is no point in keeping those tuples in the index. This strategy reduces the size of the index, which speeds up the index creation time and reduces space requirements. It also speeds up many table update operations because the index does not need to be updated in all cases [PostgreSQLv9 2018].

Partial indexes may favor better performance than complete indexes because of the smaller size of their access structures. For example, consider a complete index C and a partial index PI, both using a B+-tree on attributes A and B as the physical data structure, where the partial index has a set of restrictions R. If we have a query Q for which both C and PI are useful, it might be possible that the number of scanned blocks using PI to answer Q will be smaller than the number of scanned blocks using C. Figure 1 illustrates this case for query $Q = A > 0 \; and \; A < 10 \; and \; B =' X'$. Both indexes (complete and partial) were built with a B+-tree, and the leaf nodes represent data entries that point to physical data. In this case, the number of scanned entries using the partial index can be smaller than the number of entries scanned with the complete index. For this particular example, it is in fact – while the complete index scans 21 data entries, the partial index scans only 10 data entries to answer the query Q.

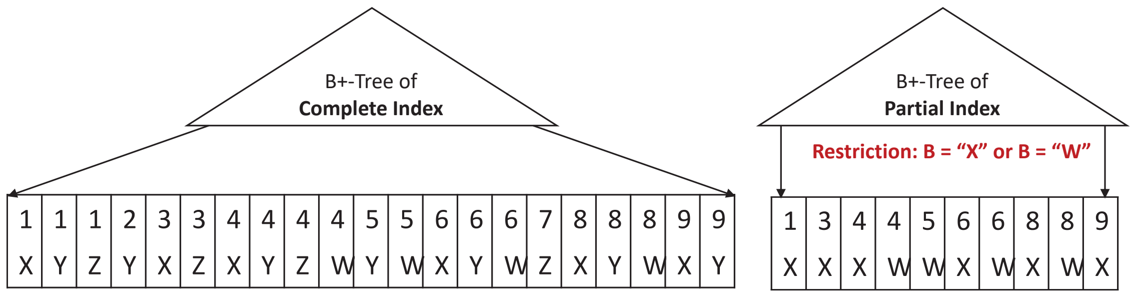

Figure 1. Representation of scanned data entries in Partial and Complete indexes for Query Q: A >0 and A<10 and B = 'X' . Values for attribute A are represented in the first row of leaf nodes, while values for attribute B are in the second row.

Finding complete indexes that may benefit a workload can be a difficult task. Besides choosing a set of attributes suitable to index creation, we also need to choose the order in which they will be indexed. There is a large number of combinations to check. The case of partial indexes is even more complex since it requires the same phases that complete indexes need and also an extra step to determine a subset of tuples which will be indexed by the partial index (the restricting selection clause).

3. Related Work

The index selection problem has been studied for many years by the database research community [Lightstone 2009]. There are two families of research in this area, which focus on development of algorithms and data structures that optimize the maintenance cost of indexes and other access structures [Labio et al. 1997], or that develop algorithms to optimize query response time [Gupta et al. 1997, Agrawal et al. 2001]. Studies about optimization of query response time may be categorized depending on how the set of candidate indexes is selected. In this work we are interest in the second family of research work [Aouiche and Darmont 2009].

Some related work about index performance focus in partial indexes as an alternative data structure. In fact, there are some cases where partial indexes are particularly useful. We discuss three situations here: (i) focus on low maintenance of the indexes; (ii) focus on saving space; and (iii) focus on reducing logical reads.

Partial indexes are useful when we have sets of tuples with a high percentage of requests and an update rate smaller than the average of updates in the table [Stonebraker 1989]. In this first situation, approaches in literature use partial indexes as a way to obtain access structures with a low maintenance cost and high performance. We may cite research work that fit to this criteria distributed in two main groups of adaptive indexing: database cracking and adaptive merging. Database cracking [Graefe and Kuno 2010a, Graefe and Kuno 2010b] combines features of automatic index and partial indexes selection by indexing results of each query. Each partitioning step creates two new sub-partitions using a logic similar to partitioning in quicksort [Idreos et al. 2011]. While database cracking functions as an incremental quicksort, with each query resulting in at most one or two partitioning steps, adaptive merging [Idreos et al. 2007a, Idreos et al. 2007b, Idreos et al. 2009, Voigt et al. 2012, Graefe et al. 2014] functions as an incremental merge-sort, with one merge step applied to all key ranges in a query result. Under adaptive merging, the first query to use a given column in a predicate produces sorted runs. Each subsequent query on that same column applies to at most one additional merge step, that only affects those key ranges that are relevant to actual queries, leaving records in all other key ranges in their initial places [Idreos et al. 2011].

The second situation covers those cases where complete indexes are very expensive due to space constraints [Seshadri and Swami 1995]. Research work that fit in this situation [Chen et al. 2011, Wu et al. 2008] exclude from the index those sets of tuples with high selectivity, or low probability to be queried. There are tuples that do not take advantage of indexes benefits and lack of interest in the current workload.

Last but not least, the third situation is the one we explore in this paper. We claim that the use of multi-column partial indexes is a good alternative to improve query performance. We remove the space constrains of the second alternative and introduce a new view of partial indexes to reduce the number of logical reads in a workload.

4. Tuning with partial indexes

We advocate that the tuning process must take into consideration solutions containing both partial and complete indexes. The selection process of partial indexes can be divided into the following phases: (i) selection of indexable attributes (ii) definition of restrictions for partial indexes and (iii) final index configuration. These steps should ensure that any partial index chosen to be part of the final access structure configuration of a database system has a high probability of being used. Moreover, it should bring actual benefits during the workload execution. Each of these phases is discussed next.

4.1. Step 1: Selection of indexable attributes

An index is a copy of values from selected columns of a table that can be searched very efficiently. It includes a low-level disk block address or a direct link to the complete row of data it was copied from. The order in which the index definition specifies the columns is important. It is possible to retrieve a set of row identifiers using only the first indexed column. However, it is not possible or efficient (on most databases) to retrieve the set of row identifiers using only the second or greater indexed column(s).

We call indexable attributes of a query the sets of attributes belonging to the same table operated by the AND operator for which it makes sense to create an index. For example, in the following query, the set of possible sets of indexable attributes is A_t = {{A},{B},{A,B}}.

```
SELECT * FROM T WHERE A > 2 and  B = 'X'
```

In order to obtain the sets of indexable attributes in a workload, we perform an automatic analysis of the predicate at the WHERE clause of each query. The analysis derives new clauses expressed as a disjunction of conjunctions, that is equivalent to the original clause. The procedure consists in modeling the operations in the WHERE clause in a tree representing the priority of operations and rotating the tree according to the equivalence of operations until no node representing an OR logic operation has an AND node as its parent. Figure 2(a) represents a clause (A OR B) AND C, where internal nodes are logic operations and leaf nodes are restrictions (predicates). Figure 2(b) represents a rotation

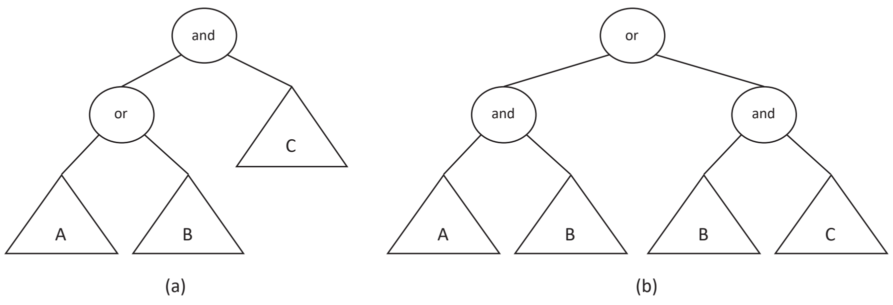

Figure 2. (a) Tree of operations of a given query; (b) Rotation of the tree in (a) so that no OR operation has an AND operation as its parent

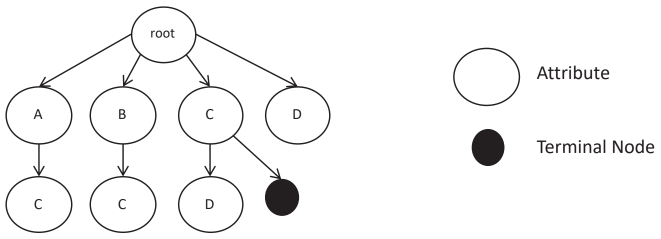

Figure 3. Trie representing a conceptual lattice

of Figure 2(a) once a node OR with a parent AND is found. We use this modified tree to obtain the indexable attributes, which in this case are $\{\{A\}, \{B\}, \{C\}, \{A, B\}, \{B, C\}\}$.

Since indexes are defined over attributes of the same table, the second step resides in separating attributes that belong to the same table. The sets of indexable attributes are obtained by looking for sets of restrictions (predicates) with attributes in the same table belonging to the same subtree with AND root node type. Once the sets of indexable attributes for each query are obtained, we may find some sets of indexable attributes repeated multiple times. These sets are those that appear more frequently in the queries of the workload, therefore, they are candidates for the creation of partial indexes.

We use a method based on conceptual lattices [Gouda and Zaki 2001] (hierarchical structure under the subset operation representing a partial order) to obtain frequent sets of indexable attributes. The data structure used for the representation of the conceptual lattices in this work is a Trie [Fredkin 1960], that counts the number of times that each set of attributes appear in the workload. Figure 3 shows a Trie representing the sets of indexable attributes $A_t = \{\{A, C\}, \{B, C\}, \{C, D\}, \{D\}\}$. In this example, if a set having three or more occurrences is considered frequent, then the attribute C must be marked as frequent and a terminal node is associated.

A set of indexable attributes is considered frequent if the proportion between the set of indexable attributes and the total number of extracted sets of indexable attributes is above a given threshold t, assuming an uniform distribution. This gives us an estimate of the number of times a given index would be useful in a query workload. The threshold t is calculated as $1/|A_t|$, where $|A_t|$ is the cardinality of the set of indexable attributes.

Once the sets of indexable attributes of each table are determined, we need a strategy to assess the impact of the candidate indexes in the workload performance. In other words, we need to identify the indexes that could be defined over the sets of indexable attributes. Hence, we adopt an heuristics of benefits to evaluate candidate indexes over sets of indexable attributes. This heuristics consists of finding, for each query of the workload, a set of candidate indexes with the higher benefit to the query according to a cost model. To do so, we obtain an aggregation of gains by estimating, for each query, the cost of running that query without the index minus the cost of executing the query with the index. Then, we obtain the benefit of that particular index by calculating the aggregated gain of the index minus its maintenance cost.

4.2. Step 2: Selection of restrictions

Let Tab be a table having a set of attributes A and a set of tuples T, where the round number of tuples fitting into a block is d. Let CI be a complete index defined over a set of indexable attributes A_t and PI a partial index defined over the same set of indexable attributes A_t having the set of restrictions RS. Each element $R \in RS$ represents a set of restrictions (predicates) relative to a subset of indexable attributes of A_t. For example, consider the definition of the partial index PI below. We can represent this partial index by a set of indexable attributes $A_t = \{A, B\}$ and the set of predicates $RS = \{\{B = 'X'\}, \{B = 'W'\}\}$.

```
CREATE INDEX PI ON T (A, B) WHERE B='X' or B='W';
```

Moreover, let Q be a query represented by a set of indexable attributes $A_q = \{a_1, a_2, ..., a_n\}$ and a set of predicates $Q_p = \{p_1, p_2, ..., p_n\}$, where each predicate p_i restricts a subset of attributes of A_q. For example, considering a table Tab with the attributes $\{A, B, C\}$ the query defined below can be represented by the sets $A_q = \{A, B\}$ and $Q_p = \{\{A > 2, B =' X'\}\}$.

```
SELECT * FROM T WHERE A > 2 and B = 'X'
```

To calculate the profit (gain) G of a partial index, we find the difference between the number of blocks scanned using the complete index CI and the number of blocks scanned using the partial index PI. Equation 1 shows a formula for estimating G by calculating the proportion of non scanned tuples to answer query Q and multiplying it by the total number of blocks in the table. In the formula, $T(C)$ denotes the number of tuples satisfying C. C can be either a query or the restrictions on an index. T is the total number of tuples in the table. For each $a_i \in A_q$, we define $CQ_i = T(X)$ and $CQ'_i = T(Y)$, such that X and Y are the set of predicates (restrictions) p_i ($X \subseteq p_i$, $Y \subseteq p_i$ and $X \neq Y$) associated to the set of indexable attributes of the set $A_t \cap a_i$ and $a_i \setminus A_t$ respectively.

$$G = \frac{(1 - \frac{T(PI)}{T}) * \frac{CQ_i}{T} * T}{d} \tag{1}$$

We then define the amount of scanned blocks B for executing the query Q (Equation 2). The notation $SB(CQ_i, CQ'_i)$ represents the amount of blocks scanned in the table when using an index.

$$B = \frac{CQ_i}{d} + SB(CQ_i, CQ'_i) \tag{2}$$

We assume in Equation 1 that the non-scanned tuples have a uniform distribution over the key A_t of the index. Moreover, the term $SB(CQ_i, CQ'_i)$ can be calculated using the results in [Mackert and Lohman 1989], that estimate the number of disk page fetches when randomly accessing k records out of n given records stored on p disk pages.

When we divide Equation 1 by Equation 2, the result quantifies how good is the benefit of using the partial index compared to the execution of Q using the complete index. Equation 3 assumes we are only interested in partial indexes having a benefit greater than a threshold w.

$$w < \frac{\dfrac{(1 - \dfrac{T(PI)}{T}) * \dfrac{CQ_i}{T} * T}{d}}{\dfrac{CQ_i}{d} + SB(CQ_i, CQ'_i)} \tag{3}$$

Manipulating Equation 3, we have that:

$$\frac{T(PI)}{T} < 1 - w * (1 + \frac{SB(CQ_i, CQ'_i) * d}{CQ_i}) \tag{4}$$

The partial index would be used only if the cost is less than a full scan. Let TB be the number of blocks occupied by the tuples of table Tab, assuming an uniform distribution:

$$\frac{T(PI)}{T} * \frac{CQ_i}{d} + SB(CQ_i, CQ'_i) < TB \tag{5}$$

Then, from Equations 4 and 5 it is possible obtain:

$$\frac{T(PI)}{T} < min(1 - w * (1 + \frac{SB(CQ_i, CQ'_i) * d}{CQ_i}), \frac{TB - SB(CQ_i, CQ'_i)}{\frac{CQ_i}{d}}) \tag{6}$$

Each restriction belonging to PI must comply to Equation 4 ensuring the partial index does reduce the number of logical reads. However, we also need to ensure that tuples in a partial index will have a high probability of being accessed. We define a similarity function $s(p_1, p_2)$ between predicates p_1 and p_2 and a ranking function *rank*(p, Q_p) for predicate p regarding the set of predicates Q_p as:

$$s(p_1, p_2) = \frac{|T(p_1) \cap T(p_2)|}{|T(p_1) \cup T(p_2)|}, \tag{7}$$

$$rank(p, Q_p) = \sum\nolimits_{p_i \in Q_p} s(p, p_i) \tag{8}$$

Then, it is possible define the restrictions of a partial index PI defined in a set of attributes A_t using the set of restrictions RS belonging to any set of indexable attributes A

in the queries of a workload, just by solving the following integer programming problem. In this program, variable a_i must have an integer value in the set $\{0\ ,\ 1\}$. This value denotes if the restriction r_i can be part of the restrictions of PI.

$$\text{maximize} \quad \sum\nolimits_{r_i \in RS} a_i * rank(r_i, RS), \text{subject to Equation 6}$$

4.3. Step 3: Final configuration

We have so far presented a methodology that finds indexable attribute sets for a given workload. These sets are candidates for the creation of either partial or complete indexes. Furthermore, let C be an indexable attribute set, w a workload and RS a set of conjunctions belonging to queries in w involving constraints on attributes in C. We present a method that may obtain a set P of conjunctions that together with those constraints in C represent a partial index noted by (C, P). The corresponding complete index (C, U) takes into account the universal set of all possible conjunctions.

However, the whole process may generate multiple partial and complete candidate indexes, as we do not consider the relationship of these indexes with the workload. We may cite as an example a table T containing attributes A, B, C and D, and query Q_1 in SQL in workload w:

```
Q1: SELECT * FROM T
    WHERE A > 1 and B > 2 and C > 3 and D > 4
```

Now, consider six possible indexable attribute sets that are frequent in w: $\{\{A\}, \{B\}, \{C\}, \{D\}, \{A, B\}, \{C, D\}\}$. These sets imply complete candidate indexes that could be used by query Q_1 separately. Thus, some of them would become redundant for this query. To avoid such a situation, we consider an algorithm that outputs all possible distinct candidate indexes that may be obtained from all candidate indexes generated in previous steps.

Let C_i be a set of indexable attributes that are frequent in a workload w and P_i a set of constraints. Function f_O computes the execution cost considering w and candidate structures in O as input. Note that the i^{th} candidate structure would be a complete index if $P_i = U$.

$$O_c = \bigcup_{i=1}^{k} < C_i, P_i >$$

Therefore, it is possible to run an algorithm that starts with an empty candidate structures set O and at each step, adds candidate structure C in O_c that together with those structures already in O can minimize the execution cost for w concerning f_O. That is, at each step we determine C in O_c that minimizes $f_O(\{< C_i, P_i >\} \cup O)$. Once available, C will be added to O whenever it increases the minimum value for f_O in the previous step.

5. Experiments

Our work intends to show that, in some cases, partial indexes may be used to improve performance rather than complete indexes. This happens because multicolumn partial indexes, with any non-categorical column in the left-most column and restrictions in any of the right-most column, can reduce the amount of logic reads in queries using the partial index when compared to the corresponding complete index.

In order to show that partial indexes may reduce the amount of logical reads, we implemented a prototype of the proposed solution in the DBX [BiobdPUC-Rio 2018] framework, whose architecture enables incorporating several database tuning solutions. Our prototype is used to generate configurations consisting of both partial and complete indexes, and also configurations consisting of only complete indexes as a result of the workload execution.

Methodology. All our experiments were run on PostgreSQL v9 DBMS, on a 64 bit computer with a 1.6GHz Quad-Core processor, 16GB of RAM and 1TB hard disk drive. We have used a database obtained from the TPC-H benchmark, configured to a scale of 100 GB. Just to mention a few examples, in this case the largest table (LineItem) has over 600 million tuples and 90MB. The supplier table has a smaller number of tuples (30,000) but uses more than 170MB, almost twice as much as LineItem.

We instantiated 8 queries out of 22 generic queries in the TPC-H benchmark. These involve simple queries in the sense that no nested queries are taken into account. We have generated 100 queries using these TPC-H queries by randomly choosing a generic query at each time. The execution of the experiment was developed in two phases. The first phase consisted of the execution of the workload using DBX to generate configurations of complete indexes only. In the second phase, DBX was tailored to generate configurations of both partial and complete indexes. This way, it was possible to determine how good partial indexes might be when compared to complete indexes. We have chosen to compare the behavior of the query execution times and also the number of logical reads for each query. It is worth noting that the time spent in query execution was measured by executing each query 10 times, discarding the first measure and computing the average of the remaining executions.

Results and discussion. The first thing we could find out about partial indexes in our research is that for single column indexes, there is no difference in the performance of query execution between partial indexes and complete indexes. This is reflected in the Equation 3 where the number of non scanned tuples for the partial index is 0.

Using the configurations suggested by the prototype implemented in DBX, we have compared the configurations of partial indexes and complete indexes, with the corresponding configuration of only complete indexes, by measuring the amount of logic reads and the execution times. Figure 4 shows the number of logic reads. The difference of logic reads in the generic queries $Q4$, $Q6$ and $Q8$ is due to the existing partial indexes. We get a reduction of logical reads of approximately 8%, 47%, and 64%, which shows that our strategy to select partial indexes have achieved good results. Note that $Q4$, $Q6$ and $Q8$ are the only queries in the workload for which partial indexes were created. This is way the amount of logic reads for the other queries are the same.

When we analyze the query execution plan for $Q8$, we note that the complete

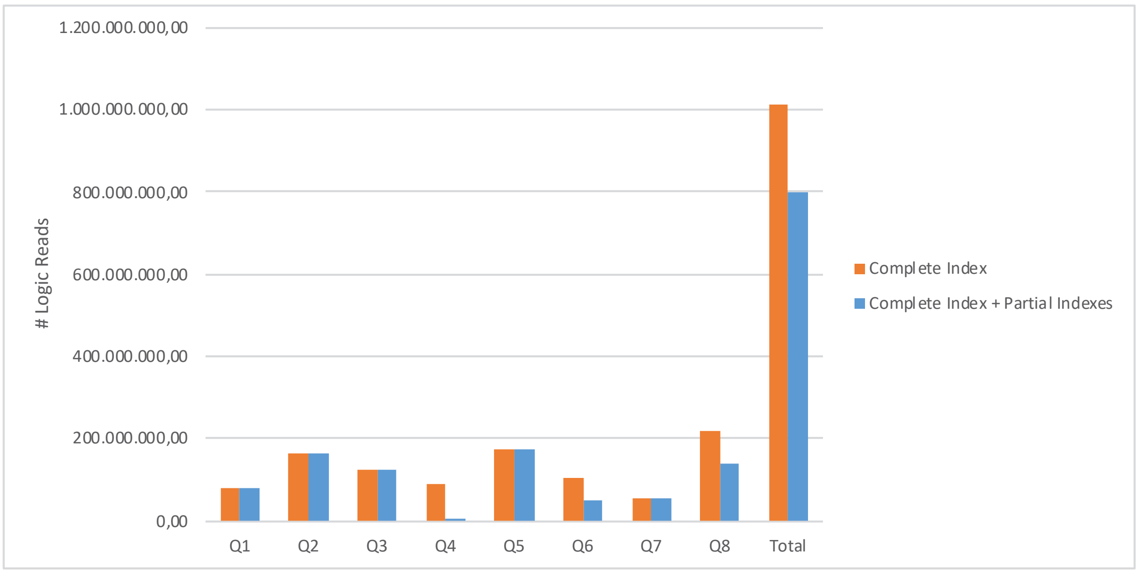

Figure 4. Logic reads by each generic query

Table 1. Hypothesis test for average difference

Query	#Instances	Complete Index + Partial Index		Complete Index		p-value
		AVG	σ	AVG	σ	
Q4	14	784.920	6.282	1.680.707	8.323	0.0001
Q6	7	1.788.024	10.371	2.595.876	13.625	0.0001
Q8	22	896.590	9.215	1.172.261	9.578	0.0001

index is not used. This happens because the partial index reduces the number of blocks to be read when compared to the complete index. A reduction in the amount of logic reads implies a reduction in the query execution time. Indeed, Figure 5 shows that the execution time for queries $Q4$, $Q6$ and $Q8$ do decrease.

The next step of our evaluation comprises checking whether the reduction on query processing time when using partial indexes for queries $Q4$, $Q6$ and $Q8$ are statistically significant. To show this, we have considered the hypothesis test of mean (or average) difference. This test is based on an attempt to reject the null hypothesis that two variables are equal. Specifically, we intend to reject the hypothesis that the execution time for queries $Q4$, $Q6$ and $Q8$ is the same for the configuration with both complete and partial indexes and the configuration with only complete indexes. Table 1 shows the results of our hypothesis test. In this table, *#Instances* refers to the number of instances of the generic query we used (please note that each instance was executed 10 times, as mentioned before). AVG is the mean of query execution time, and σ is the standard deviation. For those three queries, the difference in the mean execution time is statistically significant (*p-value* $<$ *0.05*).

6. Conclusions

This work shows that partial indexes are useful data access structures capable of improving query execution time. Particularly, we show that multi-column partial indexes can be

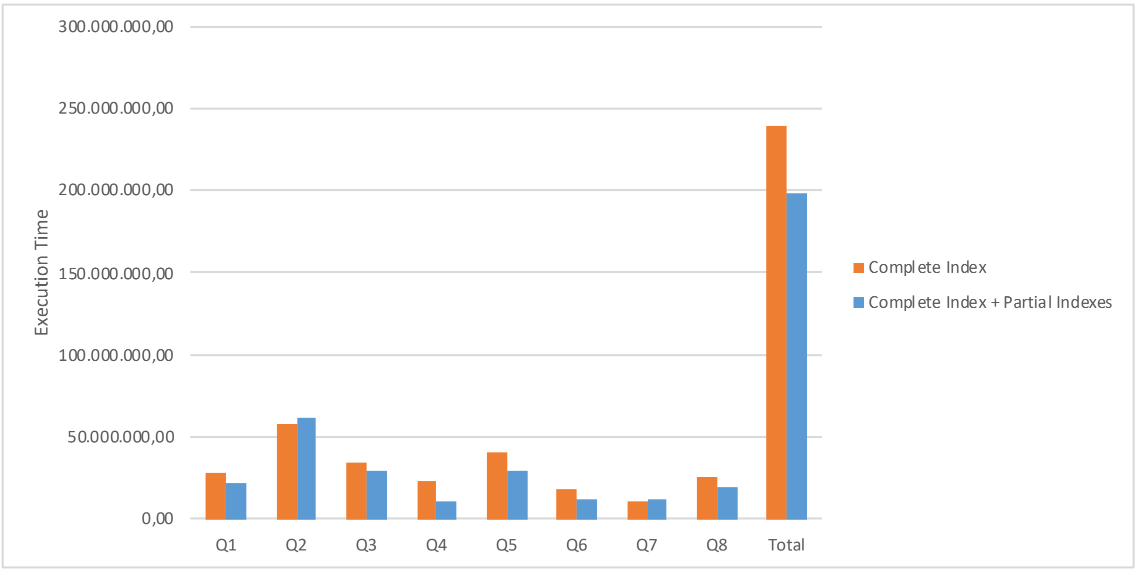

Figure 5. Execution time by each generic query

used to improve performance since it can reduce the amount of logical reads needed to process a query. We have identified at least two situations where multi-column partial indexes are effective: (i) for sets of tuples frequently queried and (ii) when there are very selective attributes. These are very common situations in practice. Therefore, it is worth counting on partial index as an additional way of obtaining efficient database systems.

Our contribution resides in an approach for database tuning that is capable of generating configurations for both partial and complete indexes. The proposed strategy is based on use cases identified for partial indexes. We have implemented and tested our proposal considering a standard OLTP benchmark (TPC-H). The obtained results are promising and show situations where partial indexes are used by the query optimizer even when the corresponding complete indexes are present. We invite the reader to check other results at [Fuentes 2016].

We intend to continue this research work by further studying the influence of the order of attributes in partial indexes. In addition, our approach could also be considered by automatic database tuning tools.

References

Agrawal, S., Chaudhuri, S., and Narasayya, V. R. (2001). Materialized view and index selection tool for microsoft SQL server 2000. In *SIGMOD Conf.*, page 608. ACM.

Aouiche, K. and Darmont, J. (2009). Data mining-based materialized view and index selection in data warehouses. *J. Intell. Inf. Syst.*, 33(1):65–93.

BiobdPUC-Rio (2018). Dbx. `https://github.com/BioBD/dbx`. [April-03-18].

Chen, C., Li, F., Ooi, B. C., and Wu, S. (2011). TI: an efficient indexing mechanism for real-time search on tweets. In *SIGMOD Conference*, pages 649–660. ACM.

Fredkin, E. (1960). Trie memory. *Commun. ACM*, 3(9):490–499.

Fuentes, A. D. (2016). Automatic fine tuning with partial indexes (in portuguese). Master's thesis, PUC-Rio Informática, Rio de Janeiro, Brasil.

Gouda, K. and Zaki, M. J. (2001). Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170. IEEE Computer Society.

Graefe, G., Halim, F., Idreos, S., Kuno, H. A., Manegold, S., and Seeger, B. (2014). Transactional support for adaptive indexing. *VLDB J.*, 23(2):303–328.

Graefe, G. and Kuno, H. A. (2010a). Adaptive indexing for relational keys. In *ICDE Workshops*, pages 69–74. IEEE Computer Society.

Graefe, G. and Kuno, H. A. (2010b). Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, Intl Conf, pages 371–381. ACM.

Gupta, H., Harinarayan, V., Rajaraman, A., and Ullman, J. D. (1997). Index selection for OLAP. In *ICDE*, pages 208–219. IEEE Computer Society.

Idreos, S., Kersten, M. L., and Manegold, S. (2007a). Database cracking. In *CIDR*, pages 68–78. www.cidrdb.org.

Idreos, S., Kersten, M. L., and Manegold, S. (2007b). Updating a cracked database. In *SIGMOD Conference*, pages 413–424. ACM.

Idreos, S., Kersten, M. L., and Manegold, S. (2009). Self-organizing tuple reconstruction in column-stores. In *SIGMOD Conference*, pages 297–308. ACM.

Idreos, S., Manegold, S., Kuno, H. A., and Graefe, G. (2011). Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597.

Labio, W., Quass, D., and Adelberg, B. (1997). Physical database design for data warehouses. In *ICDE*, pages 277–288. IEEE Computer Society.

Lightstone, S. (2009). Physical database design for relational databases. In *Encyclopedia of Database Systems*, pages 2108–2114. Springer US.

Mackert, L. F. and Lohman, G. M. (1989). Index scans using a finite LRU buffer: A validated I/O model. *ACM Trans. Database Syst.*, 14(3):401–424.

PostgreSQLv9 (2018). Partial indexes documentation. `http://www.postgresql.org/docs/9.4/static/indexes-partial.html`. [April-03-18].

Seshadri, P. and Swami, A. N. (1995). Generalized partial indexes. In *ICDE*, pages 420–427. IEEE Computer Society.

Shasha, D. E. and Bonnet, P. (2002). *Database Tuning - Principles, Experiments, and Troubleshooting Techniques*. Elsevier.

Stonebraker, M. (1989). The case for partial indexes. *SIGMOD Record*, 18(4):4–11.

Voigt, H., Jäkel, T., Kissinger, T., and Lehner, W. (2012). Adaptive index buffer. In *ICDE Workshops*, pages 308–314. IEEE Computer Society.

Wu, S., Li, J., Ooi, B. C., and Tan, K. (2008). Just-in-time query retrieval over partially indexed data on structured P2P overlays. In *SIGMOD Conf.*, pages 279–290. ACM.