Bringing Together Partitioning, Materialized Views and Indexes to Optimize Performance of Relational Data Warehouses

Ladjel Bellatreche, Michel Schneider, Hervé Lorinquer, Mukesh K. Mohania

Bringing Together Partitioning, Materialized Views and Indexes to Optimize Performance of Relational Data Warehouses

Ladjel Bellatreche[1], Michel Schneider[2], Hervé Lorinquer[2], Mukesh Mohania[3]

[1] LISI/ENSMA – Futuroscope – France bellatreche@ensma.fr
[2] LIMOS – Blaise Pascal University – France michel.schneider@isima.fr
[3]I.B.M. India Research Lab – INDIA mkmukesh@in.ibm.com

Abstract. There has been a lot of work to optimize the performance of relational data warehouses. Three major techniques can be used for this objective : enhanced index schemes (join indexes, bitmap indexes), materialized views, and data partitioning. The existing research prototypes or products use materialized views alone or indexes alone or combination of them, but none of the prototypes use all three techniques together for optimizing the performance of the relational data warehouses. In this paper we show by *a systematic experiment evaluation* that the combination of these three techniques reduces the query processing cost and the maintenance overhead significantly. We conduct several experiments and analyse the situations where the data partitioning gives better performance than the materialized views and indexes. Based on rigorous experiments, we recommend the tuning parameters for better utilization of data partitioning, join indexes and materialized views to optimize the total cost.

1 Introduction

Data warehousing technology uses the relational data schema for modeling the underlying data in a warehouse. The warehouse data can be modeled either using the star schema or the snowflake schema. In this context, OLAP queries require extensive join operations between the fact table and dimension tables [11,15]. To improve the query performance, several optimization techniques were proposed; we can cite materialized views [1,2,3,10,16], advanced indexing techniques including bitmapped indexes, join indexes (for supporting star queries), bit-sliced indexes, projection indexes [7, 8, 9, 14, 16] and data partitioning [4, 5, 12]. The data table can be fragmented into three ways: vertically, horizontally or mixed. In the context of relational warehouses, the previous studies show that horizontal partitioning is more suitable. Commercial RDBMSs like Oracle9i offer various options to use horizontal partitioning: Range, Hash, and Hybrid. This type of horizontal partitioning is called *primary horizontal partitioning* and it can be applied to dimension tables. Another type of horizontal partitioning is called derived horizontal partitioning [4]. It consists in decomposing a table based on the fragmentation schema of another table. For example, let us consider a star schema with three dimension tables (Customer, Time and Product) and a fact table Sales. The former table can be decomposed into two fact fragments $Sales_1$ and $Sales_2$ that represent all sales activities for only the male customers and all sales ac-

tivities for only the female customers, respectively. This means that the dimension table Customer is virtually partitioned using range partitioning on Gender column.

In this paper, we conduct experiments to show the effect of the combination of the three major optimization techniques by using the APB1 benchmark [13] under Oracle 9i. Along this study, the effect of updates (append and delete operations) is considered. Since the derived horizontal partitioning is not directly supported by commercial RDBMSs like Oracle9i, we present an implementation to make it operational (in this paper, we use fragmentation and partitioning interchangeably).

The paper is organized as follows: in Section 2, we introduce the necessary background and we present our explicit solution to implement the derived horizontal partitioning in commercial systems; in Section 3, we present experiments for comparing joins indexes and partitioning and we summarize the main tuning recommendations when using the three techniques; in Section 4 we present experiments for exploring the combination of the three techniques; Section 5 concludes and points some perspective.

2. Background

2.1 Benchmark

For our study, we use the dataset from the APB1 benchmark [13]. The star schema of this benchmark has one fact table Actvars and four dimension tables :

Actvars(Product_level, Customer_level, Time_level, Channel_level, UnitsSold, DollarSales, DollarCost) (24 786 000 tuples)

Prodlevel(Code_level, Class_level, Group_level, Family_level, Line_level, Division_level) (9 000 tuples)

Custlevel(Store_level, Retailer_level) (900 tuples)

Timelevel(Tid, Year_level, Quarter_level, Month_level, Week_level, Day_level) (24 tuples)

Chanlevel (Base_level, All_level) (9 tuples)

Two new attributes, week_level and day_level have been added to Timelevel table to facilitate an adequate management of updates (see section 2.3). This warehouse has been populated using the generation module of APB1. This warehouse has been installed under ORACLE 9i on a Pentium IV 1,5 Ghz microcomputer (with a memory of 256 Mo and two 7200 rps 60 Go disks) running under Windows 2000 Pro.

2.2 Workloads

The workloads used for our experiments focus on star queries. Each one has local restrictions defined in the involved dimension tables. We consider restrictions defined with predicates having only equality operator : $\boldsymbol{A = value}$, where $\boldsymbol{A}$ is an attribute name of a dimension table and $\boldsymbol{value} \in$ domain(A). When a query Q_i of a workload involves such a restriction, the workload that we consider will have $\mathbf{n_i}$ potential queries, where $\mathbf{n_i}$ represents the cardinality of domain(A). In other words, there is a potential query for each different value of domain(A). Q_i is called a *parameterized query*, and $Q_i(n_i)$ denotes its set of potential queries. For example, if we consider the parameterized query involving the predicate gender = "M", the workload will have 2 potential queries: one with the previous predicate and another with gender = "F".

2.3 Update Operations

Since the materialized views and indexes are redundant data structures, they should be maintained periodically to reflect the interactive nature of the data warehouse. The cost of maintaining materialized views, indexes and even fragments should be taken into account when combining these three techniques in order to reduce the maintenance overhead. In this study, we incorporate updates into the workload. We suppose that they occur at *regular intervals* and a certain number of queries are executed between two updates. Based on the periodicity of updates, we consider two scenarios (that keep the size of the data warehouse constant):

1. *UD (that means that Updates occur each Day)*. It consists in deleting the N oldest tuples of the fact table and inserting N new tuples (N is equal to the average activity of a day). Since our warehouse memorizes 24 786 000 tuples in the fact table for a period of 517 days, the activity of one day corresponds to an average of 47 942 tuples. So, with this scenario, 47 942 old tuples (of the fact table) are deleted and 47 942 new tuples are inserted.
2. *UW (that means that Updates occur each Week)*. This scenario is similar to the previous one, except that the periodicity of the update operations is a week. Therefore 335 594 old tuples are deleted and 335 594 new ones are inserted in the fact table.

For these scenarios, we measure the time that Oracle uses to execute all operations (updating the raw data, the materialized views, the indexes and the fragments).

2.4 Cost Models for Calculating the Total Execution Time

Let S be a set of parameterized queries $\{Q_1, Q_2, \ldots, Q_m\}$, where each Q_i has a set of potential queries ($Q_i(n_i)$). To calculate the total time to execute S and one update, we use two simple cost models called Independent_Total_Time(TI) and Proportional_Total_Time(TP). In TI, we suppose that the frequencies of the queries are equal and independent of the number of potential queries $Q_i(n_i)$. In the second one, the frequency of each Q_i is proportional to its n_i. Each model (TI and TP) will be used under the scenarios UD (where we consider TID and TPD) and UW (by considering TIW and TPW). Let t(O) be the execution time of an operation O (that can be a query, an UD or an UW). We define four cost models according to each models: TID, TIW, TPD, and TPW and defined as follows :

$TID(\alpha) = \alpha*(\Sigma_{i=1 \,..\, m}\ t(Q_i)) + t(UD)$ (1)

$TIW(\alpha) = \alpha*(\Sigma_{i=1 \,..\, m}\ t(Q_i)) + t(UW)$, where α is a positive integer, (2)

$TPD(\beta\%) = 0.01*\beta*(\Sigma_{i=1 \,..\, m}\ n_i*t(Q_i)) + t(UD)$ (3)

$TPW(\beta\%) = 0.01*\beta*(\Sigma_{i=1 \,..\, m}\ n_i*t(Q_i)) + t(UW)$ (4)

where β is an integer taken in the interval [0, 100]. In other words, $TID(\alpha)$ and $TIW(\alpha)$ give the total time needed to execute α times each query plus an update. $TPD(\beta\%)$ and $TPW(\beta\%)$ give the total time required to execute $\beta\%$ of the potential queries plus an update.

2.3 An Implementation of the Derived Horizontal Partitioning

To partition dimension tables, Oracle provides several techniques: range, hash and hybrid partitioning. The attributes used for partitioning are called *fragmentation attributes*. Oracle and the existing commercial systems do not allow partitioning of the

fact table using a fragmentation attributes belonging to a dimension table, as we will illustrate in the following example.

Suppose that we want to partition the fact table Actvars based on the virtual fragmentation schema of the dimension table ProdLevel (we assume that this former is partitioned into 4 *disjoint* segments using the attribute class_level). This will accelerate OLAP queries having restriction predicates on Class_Level. Based on this fragmentation, each tuple of a segment of the fact table will be connected to *only one* fragment of dimension ProdLevel. To achieve this goal (fragmenting the fact table based on the fragmentation schema of ProdLevel), we partition the fact table using one of the different partitioning modes (Range, Hash, Hybrid) available in the commercial systems based on the foreign key (Product_level)[1]. This fragmentation will generate 900 fact segments instead of 4 segments[2]. This example motivates the need of a mechanism that implements the derived fragmentation of the fact table. To do so, we propose the following procedure:

1- Let A = $\{A_1, ..., A_p\}$ be the set of fragmentation attributes.
2- For each A_i ($1 \leq i \leq p$) do
2.1- Add a new column (attribute) called $connect_i$ (whose domain is an integer) in the fact table. %This column gives the corresponding segment of each tuple of the fact table. For example, if the value of $connect_i$ of a given column is 1; this means that this tuple is connected (joined) to the segment 1 of the dimension table used to partition the fact table%
2.2- For each tuple of the fact table, instanciate the value of $connect_i$.
3- Specify the fragmentation of the fact table by using the attribute $connect_i$ with one of the partitioning modes (range, hash, hybrid, etc.).

To take into account the effect of data partitioning, the queries must be rewritten using the attribute $connect_i$. This implementation needs extra space (for storing the attribute(s) $connect_i$). It requires also an extra time for the update operations.

3 Comparing Derived Partitioning and Join Indexes

3.1 Queries

For this comparison, we consider separately eight queries Q1 to Q8. The queries Q1 to Q5 have one join operation and one restriction predicate. The queries Q6 to Q8 have two joins operations and two restriction predicates. Each restriction predicate has a selectivity factor. The workload and the star schema used in our experiments are given in [6] (due to the space constraint).

3.2 Experimental Results

To identify the situations where the use of derived partitioning is interesting, we have conducted three series of experiments : (1) without optimization techniques; (2) only data partitioning is used (and depends on each query), and (3) only join indexes are used. When the data partitioning is used, we have considered for each query, a number

[1] The foreign key is the single attribute that connects the fact table and the dimension table.
[2] The number of fragments of the fact table is equal the number of fragment of the dimension table.

of partitions equal the number of different values of its restriction attribute. Based on this number, we use the range mode (R) when it is small, otherwise, the hash mode (H) is used. The hybrid mode is used when queries have two join operations and two restriction predicates.

Table 1. The results for the first serie (without optimization techniques)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Query time (s)	106	61	63	53	54	61	59	56
UD (s)	68	68	68	68	68	68	68	68
UW (s)	75	75	75	75	75	75	75	75
TID(1)	174	129	131	121	122	129	127	124
TID(10)	1128	678	698	598	608	678	658	628
TPD(5%) (s)	89	105	118	269	878	214	493	2588
TPD(25%) (s)	174	251	320	1075	4118	800	2192	12668
TIW(1)	181	136	138	128	129	136	134	131
TIW(10)	1135	685	705	605	615	685	665	635
TPW(5%) (s)	96	112	125	276	885	221	500	2595
TPW(25%) (s)	181	258	327	1082	4125	807	2199	12675

When the join indexes are used, we select one index for the queries with one join and two indexes for those with two joins. For each query, we report the extra space used either by the partitioning or by the indexes, the query time, the update time for UD and UW, TID(α) and TIW(α) for two values of α (1 and 10), the values of TPD(β%) and TPW(β%)) for two values of β (5 and 25). The results are reported in the three tables Table 1, Table 2, Table 3 (one for each series).

3.3 Comments

Based on these results, the following comments are issued:

Query Time: We observe that partitioning gives a profit even for a low selectivity. The profit is very important with a high selectivity (when the number of different values for a restriction attribute is greater than 50). Join indexes give also a profit as soon as the selectivity is sufficiently high (more than 10 different values for the selection attribute). But partitioning gives better results compared to join indexes.

Update time: Join indexes are in general much more efficient than partitioning. Partitioning performs as well as indexes only for low selectivity and for daily updates. With partitioning, it is important to limit the number of partitions (less than 100 for our benchmark), otherwise the update time becomes very high. Therefore, we need to partition the fact table into *a reasonable number* of segments.

TI and TP model: It appears that partitioning is in general much more interesting than join indexes. Join indexes give better results only for high selectivity and small values of α and β (this means that the query frequency is almost the same as the update frequency).

These series of experiments summarize the following tuning recommendations when optimizing a parameterized query:

Rule 1: Data partitioning is recommended when (1) the selectivity factor of restriction predicate used in the query is low **or** (2) when the frequency of the update operation is low compare to the query frequency.
Rule 2: Join indexes are recommended when (1) the selectivity is high **or** (2) when the frequency of the update is similar to the query frequency.
In addition it is important to note that partitioning requires an additional space more significant than those required by indexes. But it remains acceptable (10% of the total space occupied by the warehouse if the number of partitions is limited to 100). This additional space is justified by the fact of adding a new column (connect) in the fact table (see Section 2.3).

Table 2.: Results for the second serie (data partitioning) (the best score for the three situations is represented in bold)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Number of partitions	4 (4R)	12 (12R)	15 (15R)	75 (75H)	300 (300H)	48 (4R*12H)	144 (12R*12H)	900 (12R*75H)
Extra space (Mo)	74	90	123	170	194	156	174	431
Query time (s)	53	9	13	4	1	11	2	1
UD (s)	69	70	70	88	112	86	105	135
UW (s)	105	92	121	154	199	144	164	220
TID(1)	**122**	**79**	**83**	**92**	113	97	107	136
TID(10)	**599**	**160**	**200**	**128**	**122**	**196**	**125**	145
TPD(5%) (s)	**80**	**75**	**80**	**103**	**127**	**112**	**119**	**180**
TPD(25%) (s)	**122**	**97**	**122**	**164**	**187**	**218**	**177**	**360**
TIW(1)	**158**	**101**	**134**	158	200	155	166	221
TIW(10)	**635**	**182**	**251**	**194**	209	**254**	**184**	230
TPW(5%) (s)	**116**	**97**	**131**	**169**	**214**	170	178	265
TPW(25%) (s)	**158**	**119**	**173**	**230**	**274**	**276**	**236**	**445**

Table 3. Results for the third serie (join indexes) (the best score for the three situations is represented in bold)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Join indexes	JI1	JI2	JI3	JI4	JI5	JI1+JI2	JI2+JI3	JI2+JI4
Extra space (Mo)	11	8	20	35	59	19	28	43
Query time (s)	152	24	57	26	8	19	9	3
UD (s)	69	69	71	69	68	68	71	69
UW (s)	98	99	101	101	104	109	110	110
TID(1)	221	93	128	95	**76**	**87**	**80**	**72**
TID(10)	1589	309	641	329	148	258	161	**99**
TPD(5%) (s)	99	83	117	168	188	114	136	204
TPD(25%) (s)	221	141	299	563	668	296	395	744
TIW(1)	250	123	158	**127**	**112**	**128**	**119**	**113**
TIW(10)	1618	339	671	361	**184**	299	200	**140**
TPW(5%) (s)	128	113	147	200	224	**155**	**175**	**245**
TPW(25%) (s)	250	171	329	595	704	337	434	785

4 Combining the Three Techniques

4.1 The queries and the cases

To evaluate the result of combining the three techniques, we conduct experiments using six SJA (Select, Join, Aggregation) queries noted Q9 to Q14. All queries are parameterized, except the query Q14. To capture the effect of the data partitioning, the number of restriction predicates in each query (except the query Q14) is equal the number of join operations.

Since a fragment is a table, the three techniques can be combined in various ways : selecting indexes and/or views on a partitioning, installing indexes and/or partitions on views, selecting views and indexes separately. We consider the following cases:

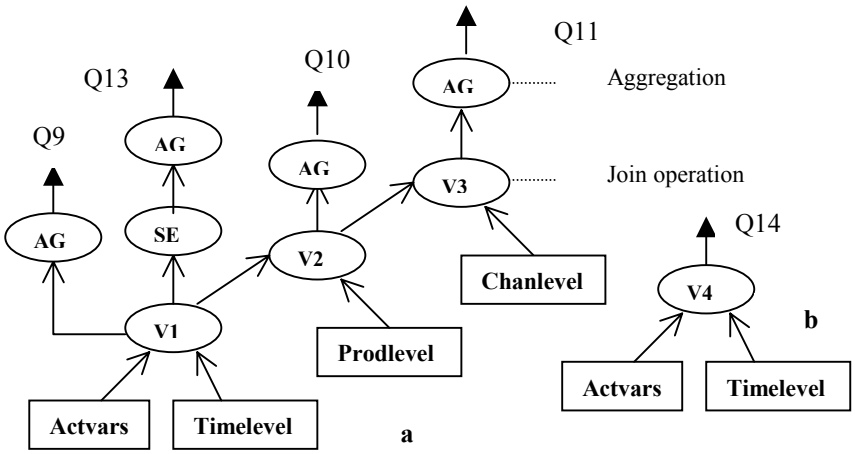

Fig. 1. The materialized views V1 to V4 used in case 2 and in case 3

Case 0 (for comparison purpose) : None optimization technique is considered;
Case 1 (to test separately the ability of a set of joins indexes) : The four join indexes JI1 on actvars (timelevel.month_level), JI2 on actvars(prodlevel family_level), JI3 on actvars(prodlevel.grouplevel), JI4 on actvars(chanlevel.all_level);
Case 2 (to situate separately the performances of a set nested materialized views) : The three nested views of figure 1a;
Case 3 (idem as the previous one with one view more) : The views of figure 1a and figure 1b;
Case 4 (to show the interest of associating a star transformation with join indexes) : The star transformation[3] with the four join indexes of case 1, plus the bitmap index incorporating the join with Timelevel, plus the view V4 of figure 1b;

[3] The star transformation is a powerful optimization technique that relies upon implicitly re-writing (or transforming) the SQL of the original star query. The end user never needs to know any of the details about the star transformation. Oracle's cost-based optimizer automatically chooses the star transformation where appropriate.

Case 5 (to evaluate the partitioning) : The derived partitioning into 96 partitions using month_level (12R) + all_level (8H) (R and H mean Range and Hash, respectively);
Case 6 (to test partitioning) : The derived partitioning into 72 partitions using all_level (9R) + family_level (8H);
Case 7 (to test partitioning) : The derived partitioning into 144 partitions using all_level (9R) + family_level (16H);
Case 8 (to test partitioning in association with indexes) : The derived partitioning into 72 partitions using all_level (9 R) + family_level (8H), plus the bitmap index on actvars(derived attribute of month_level), plus the join index JI3;
Case 9 (to test partitioning) : The derived partitioning into 96 partitions using month_level (12R) + H(family_level (8H);
Case 10 (to test partitioning in assocition with index): The derived partitioning into 96 partitions using month_level (12R) + H(family_level (8H) plus the join index JI3
Case 11 (to test partitioning in association with index): The derived partitioning into 96 partitions using month_level (12R) + H(family_level (8H), plus the join JI3, plus the view V4 of figure 5b.

4.2 Experimental Results and Comments

The results obtained for these 12 cases are reported in table 4 (ES means extra space). The execution times are given in seconds. In table 4 we found also the extra space which is needed to install the different objects (indexes, views, partitions), the values of TID(α) and TIW(α) for two values of α (1 and 10), the values of TPD(β%) and TPW(β%)) for two values of β (1 and 10).

Table 4. The results of the experiments for studying the combination of the three techniques (best score in bold, second score in grey)

Case	ES	Q9	Q10	Q11	Q12	Q13	Q14	UD	UW	TID (1)	TID (10)	TIW (1)	TIW (10)	TPD (1%)	TPD (10%)	TPW (1%)	TPW (10%)
0		61	59	65	63	58	102	65	74	473	4145	482	4154	8383	81216	8392	81225
1	158	28	5	3	29	1	102	76	138	244	1756	306	1818	600	5587	662	5649
2	301	12	1	1	77	10	103	131	292	335	2171	496	2332	-	-	-	-
3	302	12	1	1	77	10	1	147	390	249	1167	492	1410	-	-	-	-
4	167	29	5	2	20	1	1	113	236	**171**	693	**294**	**816**	494	4805	617	4928
5	279	15	10	4	31	36	97	90	145	283	2020	338	2075	2012	19445	2067	19500
6	268	78	12	1	1	71	97	98	185	358	2698	445	2785	2860	29879	2947	29966
7	297	97	6	1	1	91	126	161	217	483	3381	539	3437	3592	36628	3648	36684
8	303	46	12	1	2	1	97	122	258	281	1712	417	1848	367	4664	503	4800
9	279	16	2	1	12	35	97	90	164	253	1720	327	1794	1533	15936	1607	16010
10	305	15	2	1	12	1	97	96	176	224	1376	304	1456	**315**	**3701**	**395**	**3781**
11	305	16	2	2	12	1	1	169	485	203	**509**	519	825	468	4576	784	4892

When partitioning is used, the number of partitions should be limited to a reasonable value (less than 100 for our benchmark). With a number of partitions greater than 100, the time used for the updates becomes very high. Moreover some partitioning can disadvantage queries (for example case 6 for Q9 and Q13). Update times are highest when materialized views are used. This limits seriously the interest of the views in this kind of situations. However some configurations can give good global results despite high update times and views can be profitable in association with others techniques.

Materialized views are profitable for parameterized queries if we select a view for each value of the parameter. In general, this is not acceptable due to the huge storage and update time that we should allocated to them (materialized views). We observe also that the extra space needed to install the objects (indexes, views, partitions) remains less than 300 Mo (i.e. about 15% of the total space occupied by the warehouse).

The important observation is that join indexes alone, or materialized views alone, or partitioning alone do not provide the best results for TI or for TP. Best results are obtained when two or three techniques are combined : case 4 which combines through a star transformation the join indexes, a bitmap index and a view; case 10 which combines partitioning and a join index; case 11 which at more associates a view. Case 4 is well suited for TI but not for TP. Case 10 and 11 give good results both for TI and for TP. Combinations involving partitioning are recommended when the database administrator wants to optimize parameterized queries first.

We observe also that some configurations are good for several situations. It would be very interesting to determine such robust configurations since they remain valid after some changes in the use of the data warehouse (changes in the frequencies of queries, changes of queries, …).

5 Conclusion

The objective of this paper was to explore the possibilities of combining materialized views, indexes and partitioning in order to optimize the performances of relational data warehouses. Firstly, we have proposed an implementation of derived horizontal partitioning that allows the use of different modes of partitioning available (like range, hash and hybrid). We have compared join indexes and horizontal derived partitioning. Our results show that partitioning offers better performance (for query processing time), especially when the selectivity of the restriction predicates is low. With regard to the updates, it is less interesting, primarily when the number of partitions is high. When updates and queries interleave, a partitioning on an attribute A with n different values is advantageous as soon as a parameterized query on A is executed more than 0.05*n times between two updates. This work shows that the two techniques are rather complementary. There is thus interest to use them jointly as it had been already underlined through a theoretical study [5].

We have further compared different configurations mixing the three techniques to optimize a given set of queries. It appears that each technique used alone is not able to give the best result. Materialized views contribute in optimizing parameterized queries, but they require huge amount of storage (but they remain a good candidate for optimizing non parameterized queries). Along these experiments, two performance scenarios are distinguished: the first one is based on a star transformation with join indexes with complementary structures such as bitmap indexes or views; another one based on partitioning with complementary structures such as join indexes or views. We have noticed that the second scenario is robust since it gives good results for different situations.

Our experiments do not cover all the various uses of a warehouse. Different points should be explored in the future such as: the consideration of other types of queries (those having restrictions with OR operations, nested queries, etc.); the influence of

other kinds of updates. Nevertheless, these results allow us to list some recommendations for better tuning the warehouse: (1) the horizontal derived partitioning can play an important role in optimizing queries and the maintenance overhead, (2) incorporation of updates into the workloads may influence the selection of materialized views, indexes and data partitioning, (3) partition the fact table into a reasonable number of fragments rather having a huge segments. We think that these recommendations open the way for new algorithms for selecting simultaneously fragments, indexes and views in order to accelerate queries and optimize the maintenance overhead.

References

1. S. Agrawal, S. Chaudhuri, V.R. Narasayya, "Automated selection of materialized views and indexes in SQL databases", in Proc. 26th Int. Conf. on Very Large Data Bases (VLDB), pp. 496-505, 2000.
2. E. Baralis, S. Paraboschi, and E. Teniente, "Materialized view selection in a multidimensional database," in Proc. 23rd Int. Conf. on Very Large Data Base (VLDB), pp. 156-165, 1997.
3. L. Bellatreche, K. Karlapalem, and Q. Li, "Evaluation of indexing materialized views in data warehousing environments", in Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DAWAK), pp. 57-66, 2000.
4. L. Bellatreche, K. Karlapalem, M. Schneider and M. Mohania, "What can partitioning do for your data warehouses and data marts", in Proc. Int. Database Engineering and Application Symposium (IDEAS), pp. 437-445, 2000.
5. L. Bellatreche, M. Schneider, M. Mohania, and B. Bhargava, "Partjoin : an efficient storage and query execution design strategy for data warehousing", Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DAWAK), pp. 296-306, 2002.
6. L. Bellatreche, M. Schneider, Lorinquer, H. and M. Mohania," Bringing Together Partitioning, Materialized Views and Indexes to Optimize Performance of Relational Data Warehouses, extented version available at http://www.lisi.ensma.fr/publications.php
7. S. Chaudhuri and V. Narasayya., "An efficient cost-driven index selection tool for microsoft sql server", in Proc. Int. Conf. on Very Large Databases (VLDB), 1997, pp. 146-155.
8. C. Chee-Yong, "Indexing techniques in decision support Systems", Ph.D. Thesis, University of Wisconsin, Madison, 1999.
9. H. Gupta et al., "Index selection for olap," in Proc. Int. Conf. on Data Engineering (ICDE), pp. 208-219, 1997.
10. H. Gupta and I. S. Mumick, "Selection of views to materialize under a maintenance cost constraint," in Proc. 8th Int. Conf. on Database Theory (ICDT), pp. 453-470, 1999.
11. Informix Corporation, "Informix-online extended parallel server and informix-universal server: A new generation of decision-support indexing for enterprise data warehouses", White Paper, 1997.
12. Nicola, M. ‘’Storage Layout and I/O Performance Tuning for IBM Red Brick Data Warehouse’’, IBM DB2 Developer Domain, Informix Zone, October 2002.
13. OLAP Council, "APB-1 olap benchmark, release II", http://www.olapcouncil.org/research/bmarkly.htm.
14. P. O'Neil and D. Quass., "Improved query performance with variant indexes", in Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 38-49, 1997.
15. Red Brick Systems, "Star schema processing for complex queries", White Paper, July 1997.
16. A. Sanjay, G. Surajit, and V. R. Narasayya, "Automated selection of materialized views and indexes in microsoft sql server", in Proc. Int. Conf. on Very Large Databases (VLDB), pp. 496-505, 2000.