Some remarks on self-tuning logical database design

Fabien De Marchi[1] Mohand-Said Hacid[1]
[1]LIRIS, FRE 2672 CNRS
Univ. Claude Bernard Lyon 1
69622 Villeurbanne cedex France
{demarchi|mshacid}@liris.univ-lyon1.fr

Jean-Marc Petit[2]
[2]LIMOS, UMR 6158 CNRS
Univ. Clermont-Ferrand II
63173 Aubière, France
jmpetit@math.univ-bpclermont.fr

Abstract

Whereas self-tuning physical database design has received a lot of attention recently, self-tuning logical database design seems to be under-studied.

Roughly speaking, database administrators (DBA) have to maintain on a daily basis "efficient databases", i.e. databases for which SQL queries have to perform efficiently for end-users, while keeping "coherent" databases, i.e. databases without update problems. Moreover, the huge number of null values occurring in practice may incur a significant overhead on his daily work, either to optimize the memory layout or to maintain (or design new) SQL queries.

In order to reach a trade-off between the desire to maintain efficient databases and coherent databases, we propose a framework in which a database should be able to self-tuning its logical database schema with respect to SQL workloads and the data themselves.

We discuss the main points of this framework, its feasibility and its relationships with some data mining problems.

1. Introduction

Today's Relational DataBase Management Systems (RDBMS) require DataBase Administrators (DBA) to tune more and more parameters for an optimal use of their databases. Due to the difficulty of such a task and since a large number of companies cannot justify a full-time DBA presence, simplifying administration of RDBMS is becoming a new challenge for the database community: The idea is to have databases adjusting themselves to the characteristics of their applications [6]. For example, in the context of the *AutoAdmin* project [24], physical database tuning is investigated to improve performances of the database, e.g. index definitions or automatic statistic gathering from SQL workloads [3]. In their latest versions, the main commercial RDBMS integrate auto-administration aspects [28, 8, 3].

Whereas self-tuning physical database design has received a lot of attention recently, self-tuning logical database design seems to be under-studied.

Roughly speaking, database administrators (DBA) have to maintain on a daily basis "efficient databases", i.e. databases on which main SQL queries have to perform efficiently for end-users, while keeping "coherent" databases, i.e. databases without update problems. Moreover, the huge number of null values occurring in practice may incur a significant overhead on his daily work, either to optimize the memory layout or to maintain (or design new) SQL queries.

In order to reach a trade-off between the desire to maintain efficient databases and coherent databases, we propose in this paper a framework in which a database should be able to self-tuning (or self-restructuring) its logical database schema with respect to SQL workloads (i.e. a set of SQL queries performed over the database server during a period of time) and the data themselves. We discuss the main points of this framework, its feasibility and its relationships with some data mining problems.

This discussion is a natural extension of our previous works (see [11] for a survey), in which we proposed a project called DBA Companion devoted to the understanding of databases at the logical level. A prototype was developed [18] on top of any database running under Oracle.

Paper organization: Section 2 recalls some principles of

database design and emphasized two opposite goals. Section 3 explains the main tradeoffs to be made for logical database design with respect to either SQL workloads or the data themselves or both. Section 4 introduces the main issues we are faced with when database restructuring has to be made. Section 5 concludes the paper and sketches some perspectives of this work.

2. Motivations

Self-tuning logical database design is a necessary step when the database has to evolve for instance to better match user's requirements or when hardware/software evolution has to be performed. Many other database applications, such as database reverse engineering, data interoperability or semantic query optimization to mention a few could take advantage of logical self-tuning. Indeed, they assume that the database schema (tables + constraints) is up to date and data semantics has not been lost over the time. However, there is no guarantee at all such kind of knowledge is a priori known [9, 11].

From a database design perspective, most of existing databases have been set up using a conceptual approach for database design (for a survey on conceptual DB design, the reader is referred to [5]). Once a logical database schema has been generated from a conceptual data schema, a physical database schema has to be derived using some DBMS-dependent language. Physical database design such as data layout on external storage devices or indexes definition is out of the scope of this paper: rather, we prefer to focus on logical database design, i.e. the actual layout of attributes, tables and constraints of a database. During DB design, the translation from conceptual schema to relational schema is considered as rather straightforward. Nevertheless, de-normalization techniques may occur during this translation with respect to two main criteria:

- update problems or data redundancies, i.e. breaking the Boyce-Codd Normal Form (BCNF) or third normal form (3NF) for some schema. A huge amount of work has been done in the 80's to define normal forms with respect to functional dependencies, inclusion dependencies and multivalued dependencies (see for instance the book [1, 16] for a comprehensive survey).
- null values. The presence of NULL values can be considered as such as a conceptual database design problem [5, 23] and their occurrence are often underestimated at the database design time. In practice, null values are quite common in databases since they are very convenient to express many different kinds of incomplete information (no less than fourteen interpretations were given in [4]). Moreover, they turn out to complexify the design of SQL queries, especially when null values appear on join attributes.

Remark that the more the database is normalized, the more the lenght of *join paths* might be large (incurring costly join operations) and the less null values do appear.

3. Opposite goals

Roughly speaking, a physical database schema without data redundancy can be either an "efficient" database at the cost of having a lot of null values on join attributes or a "null free database" at the cost of performing more joins to perform equivalent SQL queries.

3.1. Maintaining coherent database

For logical database design, many options do exist within the spectrum of BCNF: different logical DB schemas may be free of data redundancy problems – i.e. they comply with BCNF wrt their FD – while different with respect to null values or join conditions.

Example 1 Let us consider two very simple databases db_1 and db_2 depicted in Table 1 and Table 2 respectively. Both are defined over the same attribute set $U = \{a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2\}$ and comply with the same constraints (functional dependencies and inclusion dependencies are given in Table 1 and 2).
Note that in our example attribute keys are underlined and foreign keys correspond to left-hand sides of inclusion dependencies.
At the conceptual level, these two databases implement a *one-to-many* binary relationship-type C with two attributes c_1, c_2 between two entity-types A and B with attributes a_1, a_2, a_3 and b_1, b_2, b_3 respectively (cf. Figure 1).
Clearly, db_1 and db_2 are equivalent with respect to functional dependencies, i.e. they are both in BCNF without lost of FD. Nevertheless, they differ as follows:

- db_1 may involve more join conditions than db_2 but **no null values** may arise on duplicated attributes a_1 and b_1 of C.
- db_2 may be more efficient than db_1, but null values occur on duplicated attributes b, c_1 and c_2 of A. Such null values can never appear in db_1 due to its database structure.

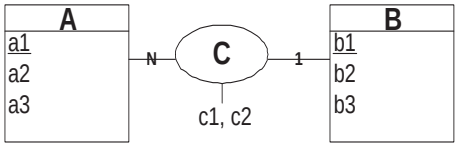

Figure 1. One to many relationship-type

A	a_1	a_2	a_3
	1	0	1
	2	0	3
	3	1	2
	4	2	5

B	b_1	b_2	b_3
	0	2	2
	1	1	1

C	a_1	b_1	c_1	c_2
	1	1	ϵ	1
	2	0	1	ϵ
	3	1	ϵ	ϵ

$I = \{C[a_1] \subseteq A[a_1], C[b_1] \subseteq B[b_1]\}$
$F = \{a_1 \rightarrow a_2a_3b_1c_1c_2, b_1 \rightarrow b_2b_3\}$.

Table 1. The database $db1$

db_2

A	a_1	a_2	a_3	b_1	c_1	c_2
	1	0	1	1	ϵ	1
	2	0	3	0	1	ϵ
	3	1	2	1	1	1
	3	1	2	ϵ	ϵ	ϵ
	4	2	5	ϵ	ϵ	ϵ

B	b_1	b_2	b_3
	0	2	2
	1	1	1

$I = \{A[b_1] \subseteq B[b_1]\}$
$F = \{a_1 \rightarrow a_2a_3b_1c_1c_2, b_1 \rightarrow b_2b_3\}$.

Table 2. The database $db2$

□

We restrict our attention on the problem of null values. Occurrences of null values are quite common in real life databases and are known to be a major difficulty for database programmers when they have to write SQL queries:

- On duplicated attributes (i.e. attributes involved in join paths): they can be a nightmare when dealing with joins, specific RDBMS functions, etc. Duplicated attributes are exactly those attributes which enable attribute-oriented models such as the relational model to simulate constructor-oriented models such as Entity-Relationship models.
- On non-duplicated attributes: Most of the time, null values were missing at the insertion time of a tuple, but such values are not used anymore to navigate through the database schema. These attributes are descriptive only, they are defined within a relation schema and convey part of the information or semantics of this relation schema. Null values on such attributes are not truly challenging for designing SQL queries.

Moreover, the presence of NULL values is very often a big issue in a knowledge discovery in database process [13], often considered by the expert during the pre-processing phase. Obviously, such treatments might have a significant impact on the discovered knowledge and on the efficiency of the data mining algorithms being used.

Let us consider the following example to show how the database structure influences the number of null values.

Example 2 From the previous example, let us assume that the size of A, B and C are as follows: 10^6 tuples in A, 10^4 in B and 10^2 in C.

If we focus on the number of null values between db_1 and db_2, we get that db_2 has much more null values than db_1 has.

Let $null_{db_i}$ be the overall number of null values occurring on $db_{db_i}, i = 1, 2$. From our example, we get the following relationship between these two numbers:

$$null_{db_2} = null_{db_1} + 3(10^6 - 100)$$

since null values on b_1, c_1, c_2 automatically appears in the relation A of db_2. In this setting, we say that db_1 is a *null free database* on duplicated attributes. □

Clearly, to get a null free database on duplicate attributes as db_1 does, the price to pay is that the length of join paths tends to be maximized, i.e. the performances of SQL queries may suffer.

As shown in the previous example, the overall number of null values in a database might be dominated by null values occurring on such attributes. We plan to conduct experiments on publicly available databases to assess this remark.

Clearly, the choice between these different solutions is difficult to be done at the database design time. A challenging issue for self-tuning databases is to make a RDBMS able to guide a DBA for self-tuning logical database schemas.

3.2. Maintaining efficient databases

In that case, there are two main options:

- Reducing length join paths without sacrificing normal form based on functional dependencies such as BCNF or third normal form (as shown in previous examples).
- Reducing length join paths by introducing data redundancy (2NF, 1NF)

3.2.1. Efficient DB without data integrity problems The aim of this alternative is to maintain BCNF or 3NF schemas while *minimizing the lenght of join paths*.

Example 3 From our previous example, the database db_2 complies with this principle. As expected, the number of relation schemas and inclusion dependencies decreases, whereas null values arise on attributes b_1, c_1, c_2 in A. □

To sum up, such kind of logical database schemas is often chosen to produce physical database schemas at DB design time, its main advantage being to minimize the length of join paths, and thus to be rather efficient. The often misunderstood problem of such schemas concerns the number of null values which can be generated once the database is operational (cf Example 2). For database designers, it might not be an important issue at database design time but that could become a nightmare for database programmers who have to devise SQL queries in presence of null values on duplicated attributes.

3.2.2. Efficient DB with data integrity problems One may be tempted to go a step beyond in order to avoid costly join operations: in that case, data integrity problems will inevitably occur due to update anomalies.

Example 4
The Table 3 shows a fully denormalized databases in which all kind of updates anomalies may occur due to data redundancy.

□

4. How to reach a compromise?

We argue that many choices made at the DB design time may be wrong once the database is operational. In order to perform the self-tuning of logical database schema, a compromise has to be reached between opposite goals.

In the spirit of [26], we argue that a good design cannot be obtained at database design time: too many parameters have to be taken into account at an early stage of the design, specifically those related to application programs accessing the database. Nevertheless, an "optimal design" could be defined and obtained with respect to the database accesses as given by SQL workloads and the data themselves.

We argue that **SQL workloads** could be used to tune the database design of operational databases since they offer a nice setting in which logical database tuning can be treated objectively - with respect to SQL workloads - instead of subjectively - with respect to the database designer expertise.

Another key information freely available to reach a good compromise is the **data** themselves. In this setting, new data mining applications arise such that key or foreign key discovery in databases [9, 11].

4.1. From SQL workloads

4.1.1. Gathering SQL workloads SQL workloads represent a set of SQL accesses performed over the database during some periods of time. They should be representative of the database activity, either Select From Where SQL queries or update SQL queries (insert/delete/update). Recently, SQL workloads can be easily gathered from operational databases by means of advanced functions available on top of major RDBMS products: a representative workload can be generated by logging activity on the server and filtering the events we want to monitor [3].

4.1.2. Using SQL statements to tune the logical database design The key idea is to tune the design with respect to three main goals: minimizing the occurrence of null values, maximizing both the efficiency of cost-sensitive SQL queries performed against the database and data integrity of the database.

Example 5 Let us consider two SQL workload scenarios for database db_2 (cf. Table 2), i.e. they represent cost-sensitive SQL queries. The first scenario leads to a workload W_1 consisting of a join between $A.b_1$ and $B.b_1$ plus some conditions on attributes $\{a_1, b_2, c_2\}$. The second scenario extends the workload W_1 with new conditions on $A.a_2$. Let W_2 be this workload.
For the database db_2, these two scenarios can be interpreted as follows:

- with W_1, the database db_1 could be used instead of db_2 with the following gain:
  - better performances of SQL queries can be expected since the number of joins does not change with respect to W_1 and the size of the involved relations could be dramatically reduced.
  - the number of null values will decrease as already explained (cf example 2).
- with W_2, the decision is more difficult since at least one SQL query of W_2 has to be rewritten with an additional join if db_1 is chosen instead of db_2 (due to the presence of a_2 in W_2).

□

4.2. From the data themselves

Data semantics in relational databases is mainly conveyed by two kind of dependencies: functional and inclusion dependencies. These two dependencies lead to the definition of key and foreign key in practice, two very popular constraints supported by most of major RDBMS software.

db_3

A	a_1	a_2	a_3	b_1	c_1	c_2	b_2	b_3
	1	0	1	1	ϵ	1	1	1
	2	0	3	0	1	ϵ	2	2
	3	1	2	1	ϵ	ϵ	1	1
	3	1	2	ϵ	ϵ	ϵ	ϵ	ϵ
	4	2	5	ϵ	ϵ	ϵ	ϵ	ϵ

$F = \{a_1 \rightarrow a_2a_3b_1c_1c_2, b_1 \rightarrow b_2b_3\}$.

Table 3. The denormalized database db_3

Nevertheless, if these constraints in a particular database have never been defined or have been lost over the time, it seems quite reasonable to have a look at the data themselves to recover them.

Example 6 Since no duplicated attributes do exist in the database db_3 in Table 3, SQL workloads are useless to cope with the logical database structure of this database whereas data mining techniques can still be applied to understand the structure of db_3. □

In this setting, challenging data mining tasks can been define to discover these dependencies from the database. Algorithms do exist for functional dependency discovery [14, 25, 19] and inclusion dependency discovery [20, 10] and despite the intrinsic complexity of these tasks, they work well for medium-size databases.

Due to the absence of constraints, inconsistencies may occur in the database and therefore *approximate dependencies*, i.e. dependencies that "almost hold" into database are worth considering. For instance, the confidence threshold of association rules is a parameter to deal with approximate association rules [2]. In the same way, approximate dependencies can be taken into account for funcionnal and inclusion dependencies [14, 25, 19, 20, 10].

From a data mining point of view, discovering FD and IND in databases is far from being realistic yet on large-size databases due to their inherent intractability. Therefore, the applicability of these propositions is questionable in a real-life setting. Nevertheless, we believe that this issue raises interesting research problems, such as the *approximation* of the two data mining problems mentioned above in such a way that the their discovery and the maintenance of the discovered FD and IND become easier, as partially done in [9].

The full integration of data-mining algorithms – or at least main data-centric steps – into query processing engine of RDBMS is also a wonderfull challenge [27, 7].

5. Issues in DB restructuring

When a restructuration has to be performed onto an operational database, one needs to define a new database schema and then to migrate the data, keeping in mind that application programs accessing the database must still be working once the new database would have been set up. We are faced with two main cases:

- Adding new constraints without changing the database schema : One needs to take care of application programs since new constraints may change the *order* in which SQL insert/update/delete statements are performed in application programs. The problems can be dealt with different options within specific options of RDBMS, as the ENABLE NOVALIDATE option under Oracle.
- Adding new constraints and changing the database schema : data migration has to be performed within the same RDBMS and applications programs have to be upgraded.

Not surprisingly, the more complicated issue arises whenever *application programs* accessing the database schema have to be upgraded.

One simple approach is to define a set of views over the new database schema to "simulate" the old database schema. Nevertheless, updating relational views is a difficult problem which is weakly supported by major RDBMS yet.

This is a major problem in practice on which we believe a lot of research remains to be done. In the sequel, we detail further the definition of constraints over existing databases.

5.1. Adding new constraints

Quite easily, one may suggest to a DBA to enforce *key* constraints (either primary key or unique and not null constraints), *foreign key* constraints, *not null* constraints or *triggers* definition for more complicated cases.

In the sequel, we will focus on foreign key, the treatment is rather similar for other constraints. Let us just recall that a foreign key is a special case of an inclusion dependency[1]:

1 Such IND are also called *key-based* inclusion dependencies (i.e. the right-hand side is a key) and are the most interesting inclusion dependencies in data modeling [21, 15, 12]. For example, they allow to represent foreign keys or isa-relationships. Furthermore, an inclusion dependency $R[X] \subseteq S[Y]$, where X is a key and Y is not a key but is a foreign key, allows modeling of cardinality constraints [12]. For an in-depth discussion on the use of inclusion dependencies in database design and the interaction between functional dependencies and inclusion dependencies, the reader is referred to [21, 22, 17, 16].

X is a foreign key of R (to Y of S) iff $R[X] \subseteq S[Y]$ holds and Y is a key of S.

Clearly, a discovered inclusion dependency – exact or approximate – may indicate a missing foreign key or a schema misconception. Once a sequence of attributes is known to be a foreign key, it could be enforced to prevent invalid data entry into the database.

More precisely, given an IND $R[X] \subseteq S[Y]$, three cases are relevant for self-tuning logical database:

- Y is a key of S: a *foreign key* can be defined on X of R.
- Y is not a key but is a foreign key of S: a *trigger* can be defined to enforce this constraint.
- Y is neither a key nor a foreign key of S: a *normalization* can be performed on the database, i.e. the database structure may be changed.

In addition, the discovered foreign keys can be used to improve performance of joins over multiple relations. Indeed, foreign keys constitute good candidates for indexing or for creating clusters.

6. Conclusion

Considerations introduced in this paper for self-tuning the logical database design are simple though very important in practice. Between the desire to maintain efficient databases for end-users and the desire to maintain coherent databases for database programmers and end-users, we have pointed out how SQL workloads and the data themselves could be used to reach a compromise among contradictory objectives.

Through the notion of *assistant*[2], one could envision the self-tuning of logical database design from both SQL workloads and the data themselves.

Along the paper, we have pointed out some challenges that remain to be addressed to cope with self-tuning logical database design.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Fondements des bases de donnes*. Addison Wesley, 2000.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the* 20^{th} *International Conference on Very Large Databases, Santiago de Chile, Chile*, pages 487–499, 1994.

[3] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft SQL Server 2005. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *International Conference on Very Large Data Bases, Toronto, Canada*, pages 1110–1121. Morgan Kaufmann, 2004.

[4] ANSI/X3/SPARC. Interim report: Ansi/x3/sparc study group on data base management systems. *FDT - Bulletin of ACM SIGMOD*, 7(2):1–140, 1975.

[5] C. Batini, S. Ceri, and S. Navathe. *Conceptual Database Design: an Entity-Relationship Approach*. Benjamin Cummings, 1992.

[6] P. Bernstein and al. The ASILOMAR report on database research. *ACM Sigmod Record*, 27(4):74–80, 1998.

[7] S. Chaudhuri. Data mining and database systems: Where is the intersection? *Data Engineering Bulletin*, 21(1):4–8, 1998.

[8] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL tuning in ORACLE 10g. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *International Conference on Very Large Data Bases, Toronto, Canada*, pages 1098–1109. Morgan Kaufmann, 2004.

[9] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *ACM SIGMOD Conference 2002*, pages 240–251, Madison, Wisconsin, USA, 2002.

[10] F. De Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Proceedings of the* 7^{th} *International Conference on Extending Database Technology*, volume 2287 of *Lecture Notes in Computer Science*, pages 464–476, Prague, Czech Republic, 2002. Springer-Verlag.

[11] F. De Marchi, S. Lopes, J.-M. Petit, and F. Toumani. Analysis of existing databases at the logical level: the dba companion project. *ACM Sigmod Record*, 32(1):47–52, 2003.

[12] C. Fahrner and G. Vossen. A Survey of Database Design Transformations Based on the Entity-Relationship Model. *Data and Knowledge Engineering*, 15:213–250, 1995.

[13] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, first edition, 2000.

[14] Y. Huhtala, J. Krkkinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(3):100–111, 1999.

[15] P. Johannesson. A Method for Transforming Relational Schemas into Conceptual Schemas. In *Proceedings of the* 10^{th} *IEEE International Conference on Data Engineering*, pages 190–201, Houston, Texas, February 1994. IEEE Computer Society.

[16] M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999.

[17] M. Levene and G. Loizou. Guaranteeing no interaction between functional dependencies and tree-like inclusion dependencies. *Theoritical Computer Science*, 254(1-2):683–690, 2001.

[18] S. Lopes, F. De Marchi, and J.-M. Petit. DBA companion: A tool for logical database tuning (demo). In 20^{th} *Proceedings of the IEEE International Conference on Data Engineering*, page 859, Boston, USA, 2004. IEEE Computer Society.

2 Assistants or advisor tools are now very common as an aid for main DBA tasks such as database set up, index tuning ...

[19] S. Lopes, J.-M. Petit, and L. Lakhal. Functional and approximate dependencies mining: Databases and FCA point of view. *Special issue of Journal of Experimental and Theoretical Artificial Intelligence*, 14(2/3):93–114, 2002.

[20] S. Lopes, J.-M. Petit, and F. Toumani. Discovering interesting inclusion dependencies: Application to logical database tuning. *Information Systems*, 17(1):1–19, 2002.

[21] H. Mannila and K.-J. Räihä. Design by example: An application of Armstrong relations. *Journal of Computer and System Sciences*, 33(2):126–141, 1986.

[22] H. Mannila and K.-J. Räihä. Algorithms for Inferring Functional Dependencies from Relations. *Data and Knowledge Engineering*, 12:83–99, 1994.

[23] H. Mannila and K.-J. Räihä. *The Design of Relational Databases*. Addison-Wesley, second edition, 1994.

[24] Microsoft. , `http://www.research.microsoft.com/dmx/autoadmin`.

[25] N. Novelli and R. Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the International Conference on Database Theory, London, UK*, volume 1973 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, 2001.

[26] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Mc Graw-Hill, third ed. edition, 2003.

[27] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating mining with relational database systems: Alternatives and implications. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 343–354. ACM Press, 1998.

[28] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated automatic physical database design. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *International Conference on Very Large Data Bases, Toronto, Canada*, pages 1087–1097. Morgan Kaufmann, 2004.