The Design of DWEB

Jérôme Darmont, Fadila Bentayeb, and Omar Boussaïd
ERIC, University of Lyon 2
5 av. Pierre Mendès-France
69676 Bron Cedex
France
{jdarmont|boussaid|bentayeb}@eric.univ-lyon2.fr

Abstract

Data warehouse architectural choices and optimization techniques are critical to decision support query performance. To facilitate these choices, the performance of the designed data warehouse must be assessed. This is usually done with the help of benchmarks, which can either help system users comparing the performances of different systems, or help system engineers testing the effect of various design choices. While the TPC standard decision support benchmarks address the first point, they are not tuneable enough to address the second one and fail to model different data warehouse schemas. By contrast, our Data Warehouse Engineering Benchmark (DWEB) allows to generate various ad-hoc synthetic data warehouses and workloads. DWEB is fully parameterized to fulfill data warehouse design needs. However, two levels of parameterization keep it relatively easy to tune. Finally, DWEB is implemented as a Java free software that can be interfaced with most existing relational database management systems. A sample usage of DWEB is also provided in this paper.

Keywords: Data warehouses, decision support queries, OLAP, benchmarking, performance evaluation, data warehouse design.

1 Introduction

When designing a data warehouse, choosing an architecture is crucial. Since it is very dependant on the domain of application and the analysis objectives that are selected for decision support, different solutions are possible. In the ROLAP (Relational OLAP) environment we consider, the most popular solutions are by far star, snowflake, and constellation schemas [Inm02, KR02], and other modeling possibilities might exist. This choice of architecture is not neutral: it always has advantages and drawbacks and greatly influences the response time of decision support queries. For example, a snowflake schema with hierarchical dimensions improves analysis power, but induces many more costly join operations than a star schema. Once the architecture is selected, various optimization techniques such as indexing or materialized views further influence querying and refreshing performance. Again, it is a matter of trade-off between the improvement brought by a given technique and its overhead in terms of maintenance time and additional disk space; and also between different optimization techniques that may cohabit.

To help users make these critical choices of architecture and optimization techniques, the performance of the designed data warehouse needs to be assessed. However, evaluating data warehousing and decision support technologies is an intricate task. Though pertinent, general advice is available, notably on-line [Pen03, Gre04a], more quantitative elements regarding sheer performance are scarce. Thus, we propose in this paper a data warehouse benchmark we named DWEB (the *Data Warehouse Engineering Benchmark*). A benchmark may be defined as a synthetic database model and a workload model. Different goals may be achieved by using a benchmark:

1. compare the performances of various systems in a given set of experimental conditions (users);

2. evaluate the impact of architectural choices or optimisation techniques on the performances of one given system (system designers).

The Transaction Processing Performance Council (TPC) [Tra04], a non-profit organization, defines standard benchmarks and publishes objective and verifiable performance evaluations to the industry. These benchmarks mainly aim at the first benchmarking goal we identified. However, these benchmarks only model one fixed type of database and they are not very tuneable: the only parameter that defines their database is a scale factor determining its size. Nethertheless, in a development context, it may be interesting to test a solution (an indexing strategy, for instance) using various database configurations. Furthermore, though there is an ongoing effort at the TPC to design a data warehouse benchmark, the current TPC decision support benchmarks do not properly model a data warehouse. They do not address specific warehousing issues such as the ETL (Extract, Transform, Load) process or OLAP (On-Line Analytical Processing) querying either.

Hence, the aim of this paper is to present an operational benchmark for data warehouses. More precisely, our objective is to design a benchmark that helps generating ad-hoc synthetic data warehouses (modeled as star, snowflake, or constellation schemas) and workloads, mainly for engineering needs (second benchmarking objective). It is indeed very important to achieve the different kinds of schemas that are used in data warehouses, and to allow users to select the precise architecture they need to evaluate.

The remainder of this paper is organized as follows. First, we discuss the state of the art regarding decision support benchmarks in Section 2. We motivate the need for a data warehouse benchmark in Section 3. Then we detail DWEB's database and workload in Sections 4 and 5, respectively. We also briefly present our implementation of DWEB in Section 6. We finally illustrate how our benchmark can be used in Section 7. We conclude this paper and provide future research directions in Section 8.

2 Existing decision support benchmarks

To the best of our knowledge, very few decision support benchmarks have been designed out of the TPC. Some do exist [Dem95], but their specification is not fully published. Some others are not available any more, such as the OLAP APB-1 benchmark that was issued in the late nineties by the OLAP council, an organization that does not seem to exist any more. This is why we focus on the TPC benchmarks in this section.

TPC-D [Bal93, Bha96, Tra98] appeared in the mid-nineties, and forms the base of TPC-H and TPC-R that have now replaced it [PF00, Tra03a, Tra03b]. TPC-H and TPC-R are actually identical, only their usage varies. TPC-H is for ad-hoc querying (queries are not known in advance and optimizations are forbidden), while TPC-R is for reporting (queries are known in advance and optimizations are allowed). TPC-H and TPC-R exploit the same relational database schema as TPC-D: a classical *product-order-supplier* model (represented as a UML class diagram in Figure 1); and the workload from TPC-D supplemented with five new queries.

This workload is constituted of:

- twenty-two SQL-92 parameterized, decision-oriented queries labeled Q1 to Q22;

- two refresh functions RF1 and RF2 that essentially insert and delete tuples in the ORDER and LINEITEM tables.

The query parameters are subtituted with the help of a random function following a uniform distribution. Finally, the protocol for running TPC-H or TPC-R includes:

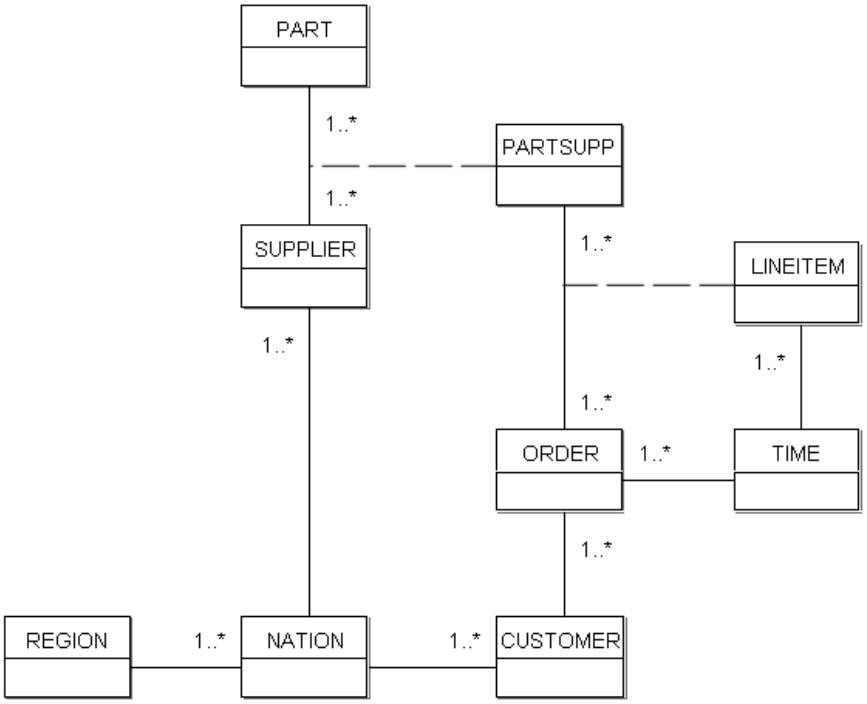

Figure 1: TPC-D, TPC-H, and TPC-R database schema

1. a load test;
2. a performance test (executed twice) further subdivided into:
   - a power test,
   - a throughput test.

Three primary metrics describe the results in terms of power, throughput, and a composition of the two.

TPC-DS [PSKL02], which is currently under development, is the designated successor of TPC-H and TPC-R, and more clearly models a data warehouse. TPC-DS' database schema, whose fact tables are represented in Figure 2, models the decision support functions of a retail product supplier as several snowflake schemas. This model also includes fifteen dimensions that are shared by the fact tables. Thus, the whole model is a constellation schema.

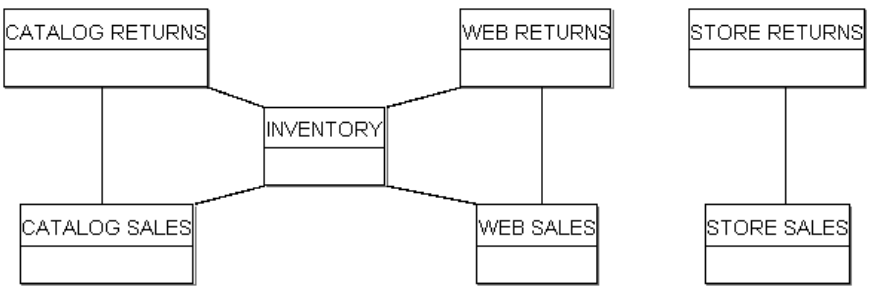

Figure 2: TPC-DS data warehouse schema

TPC-DS' workload is made of four classes of queries:

- reporting queries,
- ad-hoc decision support queries,
- interactive OLAP queries,
- data extraction queries.

A set of about five hundred queries is generated from query templates written in SQL-99 (with OLAP extensions). Substitutions on the templates are operated using non-uniform random distributions. The data warehouse maintenance process includes a full ETL process and a specific treatment of the dimensions. For instance, historical dimensions preserve history as new dimension entries are added, while non-historical dimensions do not keep aged data. Finally, the execution model of TPC-DS consists of four steps:

1. a load test,
2. a query run,
3. a data maintenance run,
4. another query run.

A single throughput metric is proposed, which takes the query and maintenance runs into account.

3 Motivation

Our first motivation to design a data warehouse benchmark is that we need one to evaluate the efficiency of performance optimization techniques (such as autoindexing techniques) we are currently developping. To the best of our knowledge, no such benchmark has been published yet. TPC-H and TPC-R's database schema, which is inherited from the older and obsolete benchmark TPC-D, is not a data warehouse schema such as the typical star schema and its derivatives. Furthermore, their workload, though decision-oriented, does not include explicit OLAP queries either. These benchmarks are indeed implicitely considered obsolete by the TPC that has issued some specifications for their successor: TPC-DS. However, TPC-DS has been under development for two years now and is not completed yet. This might be because of its high complexity, especially at the ETL and workload levels.

Furthermore, although the TPC decision support benchmarks are scaleable according to Gray's definition [Gra93], their schema is fixed. For instance, TPC-DS' constellation schema cannot easily be simplified into a simple star schema. It must be used "as is". Different ad-hoc configurations are not possible. Furthermore, there is only one parameter to define the database, the Scale Factor (SF), which sets up its size (from 1 to 100,000 GB). The user cannot control the size of the dimensions and the fact tables separately, for instance. Finally, the workload is not tuneable at all: the number of generated queries directly depends on SF in TPC-DS, for example. The user has no control on the workload's definition.

Finally, in a context where data warehouse architectures and decision support workloads depend a lot on the domain of application, it is very important that users who wish to evaluate the impact of architectural choices or optimisation techniques on global performance can choose and/or compare between several configurations. The TPC benchmarks, which aim at standardized results, are not well adapted to this purpose.

For all these reasons, we decided to design a full data warehouse benchmark that would be able to model various configurations of database and workload, while being simpler to develop than TPC-DS.

4 DWEB database

4.1 Schema

Our design objective for DWEB is to be able to model the different kinds of data warehouse architectures that are popular within a ROLAP environment:

- classical star schemas,
- snowflake schemas with hierarchical dimensions,
- constellation schemas with multiple fact tables and shared dimensions.

To achieve this goal, we propose a data warehouse metamodel (represented as a UML class diagram in Figure 3) that can be instantiated into these different schemas. We view this metamodel as a middle ground between the multidimensional metamodel from the Common Warehouse Metamodel (CWM) [Obj03, PCTM03] and the eventual benchmark model. Our metamodel is actually an instance of the CWM metamodel, which could be qualified as a meta-metamodel in our context.

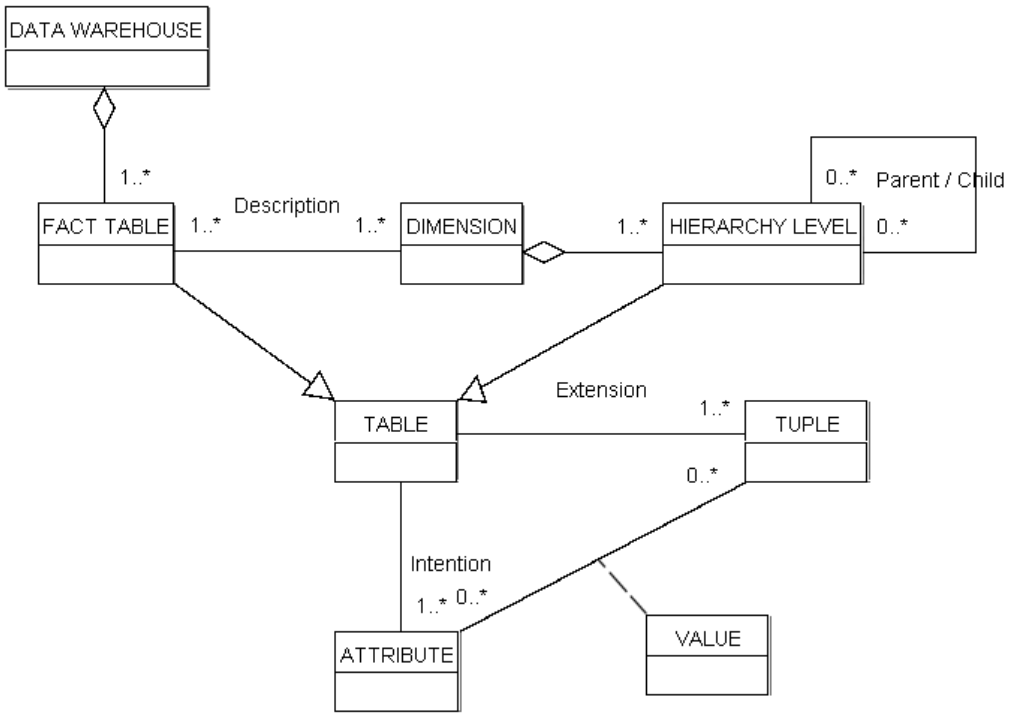

Figure 3: DWEB data warehouse metaschema

Our metamodel is quite simple, but it is sufficient to model the data warehouse schemas we aim at (star, snowflake, and constellation schemas). Its upper part describes a data warehouse (or a datamart, if a datamart is viewed as a small, dedicated data warehouse) as constituted of one or several fact tables that are each described by several dimensions. Each dimension may also describe several fact tables (shared dimensions). Each dimension may be constituted of one or several hierarchies made of different levels. There can be only one level if the dimension is not a hierarchy.

Both fact tables and dimension hierarchy levels are relational tables, which are modeled in the lower part of Figure 3. Classically, a table or relation is defined in intention by its attributes and in extension

by its tuples or rows. At the intersection of a given attribute and a given tuple lies the value of this attribute in this tuple.

4.2 Parameterization

DWEB's database parameters help users selecting the data warehouse architecture they need in a given context.

The main difficulty in producing a data warehouse schema is parameterizing the instantiation of the metaschema. We indeed try to meet the four key criteria that make a "good" benchmark, as defined by Gray [Gra93]:

- *relevance:* the benchmark must answer our engineering needs (as expressed in Section 1);
- *portability:* the benchmark must be easy to implement on different systems;
- *scalability:* it must be possible to benchmark small and large databases, and to scale up the benchmark;
- *simplicity:* the benchmark must be understandable, otherwise it will not be credible nor used.

Relevance and simplicity are clearly two orthogonal goals. Introducing too few parameters reduces the model's expressiveness, while introducing too many parameters makes it difficult to apprehend by potential users. Furthermore, few of these parameters are likely to be used in practice. In parallel, the generation complexity of the instantiated schema must be mastered. To solve this dilemna, we capitalize on the experience of designing the OCB object-oriented database benchmark [DS00]. OCB is generic and able to model all the other existing object-oriented database benchmarks, but it is controlled by far too many parameters, few of which are used in practice. Hence, we propose to divide the parameter set into two subsets.

- The first subset of so-called low-level parameters allows an advanced user to control everything about the data warehouse generation (Table 1). However, the number of low-level parameters can increase dramatically when the schema gets larger. For instance, if there are several fact tables, all their characteristics, including dimensions and their own characteristics, must be defined for each fact table.
- Thus, we designed a layer above with much fewer parameters that may be easily understood and set up (Table 2). More precisely, these high-level parameters are average values for the low-level parameters. At database generation time, the high-level parameters are exploited by random functions (following a gaussian distribution) to automatically set up the low-level parameters. Finally, unlike the number of low-level parameters, the number of high-level parameters always remains constant and reasonable (less than ten parameters).

Users may choose to set up either the full set of low-level parameters, or only the high-level parameters, for which we propose default values that correspond to a snowflake schema. Note that these parameters control both schema and data generation.

Remarks:

- Since shared dimensions are possible, $TOT\_NB\_DIM \leq \sum_{i=1}^{NB\_FT} NB\_DIM(i)$.
- The cardinal of a fact table is usually lower or equal to the product of its dimensions' cardinals. This is why we introduce the notion of density. A density rate of one indicates that all the possible combinations of the dimension primary keys are present in the fact table. When the density rate

Parameter name	Meaning
NB_FT	Number of fact tables
$NB\_DIM(f)$	Number of dimensions describing fact table $\#f$
TOT_NB_DIM	Total number of dimensions
$NB\_MEAS(f)$	Number of measures in fact table $\#f$
$DENSITY(f)$	Density rate in fact table $\#f$
$NB\_LEVELS(d)$	Number of hierarchy levels in dimension $\#d$
$NB\_ATT(d,h)$	Number of attributes in hierarchy level $\#h$ of dimension $\#d$
$HHLEVEL\_SIZE(d)$	Number of tuples in the highest hierarchy level of dimension $\#d$
$DIM\_SFACTOR(d)$	Size scale factor in the hierarchy levels of dimension $\#d$

Table 1: DWEB warehouse low-level parameters

Parameter name	Meaning	Default value
AVG_NB_FT	Average number of fact tables	1
AVG_NB_DIM	Average number of dimensions per fact table	5
$AVG\_TOT\_NB\_DIM$	Average total number of dimensions	5
AVG_NB_MEAS	Average number of measures in fact tables	5
$AVG\_DENSITY$	Average density rate in fact tables	0.6
AVG_NB_LEVELS	Average number of hierarchy levels in dimensions	3
AVG_NB_ATT	Average number of attributes in hierarchy levels	5
$AVG\_HHLEVEL\_SIZE$	Average number of tuples in highest hierarchy levels	10
$DIM\_SFACTOR$	Average size scale factor within hierarchy levels	10

Table 2: DWEB warehouse high-level parameters

decreases, we progressively eliminate some of these combinations (see Section 4.3). This parameter helps controlling the size of the fact table, independantly of the size of its dimensions, which are defined by the $HHLEVEL\_SIZE$ and $DIM\_SFACTOR$ parameters (see below).

- Within a dimension, a given hierarchy level normally has a greater cardinality than the next level. For example, in a *town-region-country* hierarchy, the number of towns must be greater than the number of regions, which must be in turn greater than the number of countries. Furthermore, there is often a significant scale factor between these cardinalities (e.g., one thousand towns, one hundred regions, ten countries). Hence, we model the cardinality of hierarchy levels by assigning a "starting" cardinality to the highest level in the hierarchy ($HHLEVEL\_SIZE$), and then by multiplying it by a predefined scale factor ($DIM\_SFACTOR$) for each lower-level hierarchy.
- The global size of the data warehouse is assessed at generation time (see Section 6) so that the user retains full control over it.

4.3 Generation algorithm

The instantiation of the DWEB metaschema into an actual benchmark schema is done in two steps:

1. build the dimensions;
2. build the fact tables.

The pseudo-code for these two steps is provided in Figures 4 and 5, respectively. Each of these steps is further subdivided, for each dimension or each fact table, into generating its intention and extension. In addition, hierarchies of dimensions must be managed. Note that they are generated starting from the highest level of hierarchy. For instance, for our *town-region-country* sample hierarchy, we build the country level first, then the region level, and eventually the town level. Hence, tuples from a given

hierarchy level can refer to tuples from the next level (that are already created) with the help of a foreign key.

```
For i = 1 to TOT_NB_DIM do
    previous_ptr = NIL
    size = HHLEVEL_SIZE(i)
    For j = 1 to NB_LEVELS(i) do
       // Intention
       hl = New(Hierarchy_level)
       hl.intention = Primary_key()
       For k = 1 to NB_ATT(i,j) do
          hl.intention = hl.intention ∪ String_descriptor()
       End for
       // Hierarchy management
       hl.child = previous_ptr
       hl.parent = NIL
       If previous_ptr ≠ NIL then
          previous_ptr.parent = hl
          hl.intention = hl.intention
             ∪ previous_ptr.intention.primary_key // Foreign key
       End if
       // Extension
       hl.extension = ∅
       For k = 1 to size do
          new_tuple = Integer_primary_key()
          For l = 1 to NB_ATT(i,j) do
             new_tuple = new_tuple ∪ Random_string()
          End for
          If previous_ptr ≠ NIL then
             new_tuple = new_tuple ∪ Random_key(previous_ptr)
          End if
          hl.extension = hl.extension ∪ new_tuple
       End for
       previous_ptr = hl
       size = size * DIM_SFACTOR(i)
    End for
    dim(i) = hl // First (lowest) level of the hierarchy
End for
```

Figure 4: DWEB dimensions generation algorithm

We use three main classes of functions and one procedure in these algorithms.

1. `Primary_key()`, `String_descriptor()` and `Float_measure()` return attribute names for primary keys, descriptors in hierarchy levels, and measures in fact tables, respectively. These names are labeled sequentially and prefixed by the table's name (e.g., DIM1_1_DESCR1, DIM1_1_DESCR2...).

2. `Integer_primary_key()`, `Random_key()`, `Random_string()` and `Random_float()` return sequential integers with respect to a given table (no duplicates are allowed), random instances of the specified table's primary key (random values for a foreign key), random strings of fixed size (20 characters) selected from a precomputed referential of strings and prefixed by the corresponding attribute name, and random single-precision real numbers, respectively.

3. `Random_dimension()` returns a dimension that is chosen among the existing dimensions that are not already describing the fact table in parameter.

4. `Random_delete()` deletes one tuple at random from the extension of a table.

```
For i = 1 to NB_FT do
    // Intention
    ft(i).intention = ∅
    For j = 1 to NB_DIM(i) do
       j = Random_dimension(ft(i))
       ft(i).intention = ft(i).intention ∪ ft(i).dim(j).primary_key
    End for
    For j = 1 to NB_MEAS(i) do
       ft(i).intention = ft(i).intention ∪ Float_measure()
    End for
    // Extension
    ft(i).extension = ∅
    For j = 1 to NB_DIM(i) do // Cartesian product
       ft(i).extension = ft(i).extension × ft(i).dim(j).primary_key
    End for
    to_delete = DENSITY(i) * |ft(i).extension|
    For j = 1 to to_delete do
       Random_delete(ft(i).extension)
    End for
    For j = 1 to |ft(i).extension| do // With |ft(i).extension| updated
       For k = 1 to NB_MEAS(i) do
          ft(i).extension.tuple(j).measure(k) = Random_float()
       End for
    End for
End for
```

Figure 5: DWEB fact tables generation algorithm

Except in the `Random_delete()` procedure, where the random distribution is uniform, we use gaussian random distributions to indroduce a skew, so that some of the data, whether in the fact tables or the dimensions, are referenced more frequently than others as it is normally the case in real-life data warehouses.

Remark: The way density is managed in Figure 5 is grossly non-optimal. We chose to present the algorithm that way for the sake of clarity, but the actual implementation does not create all the tuples from the cartesian product, and then delete some of them. It directly generates the right number of tuples by using the density rate as a probability for each tuple to be created.

5 DWEB workload

In a data warehouse benchmark, the workload may be subdivided into:

- a load of decision support queries (mostly OLAP queries);
- the ETL (data generation and maintenance) process.

To design DWEB's workload, we inspire both from TPC-DS' workload definition (which is very elaborate) and information regarding data warehouse performance from other sources [BMC00, Gre04b]. However, TPC-DS' workload is quite complex and somehow confusing. The reporting, ad-hoc decision support and OLAP query classes are very similar, for instance, but none of them include any specific OLAP operator such as Cube or Rollup. Since we want to meet Gray's simplicity criterion, we propose a simpler workload. Furthermore, we also have to design a workload that is consistent with the variable nature of the DWEB data warehouses.

We also, in a first step, mainly focus on the definition of a query model. Modeling the full ETL process is a complex task that we postpone for now. We consider that the current DWEB specifications

provide a raw loading evaluation framework. The DWEB database may indeed be generated into flat files, and then loaded into a data warehouse using the ETL tools provided by the system.

5.1 Query model

The DWEB workload models two different classes of queries:

- purely decision-oriented queries involving common OLAP operations, such as cube, roll-up, drill down and slice and dice;
- extraction queries (simple join queries).

We define our generic query model (Figure 6) as a grammar that is a subset of the SQL-99 standard, which introduces much-needed analytical capabilities to relational database querying. This increases the ability to perform dynamic, analytic SQL queries.

Query ::-	
Select	![<Attribute Clause> | <Aggregate Clause> | [<Attribute Clause>, <Aggregate Clause>]]
From	!<Table Clause> [<Where Clause> ‖ [<Group by Clause> *<Having Clause>]]
Attribute Clause ::-	*Attribute Name* [[, <Attribute Clause>] | ⊥]
Aggregate Clause ::-	![*Aggregate Function Name* (*Attribute Name*)] [**As** *Alias*] [[, <Aggregate Clause>] | ⊥]
Table Clause ::-	*Table Name* [[, <Table Clause>] |⊥]
Where Clause ::-	**Where** ![<Condition Clause> | <Join Clause> | [<Condition Clause> **And** <Join Clause>]]
Condition Clause ::-	![*Attribute Name* <Comparison Operator> <Operand Clause>] [[<Logical Operator> <Condition Clause>] | ⊥]
Operand Clause ::-	[*Attribute Name* | *Attribute Value* | *Attribute Value List*]
Join Clause ::-	![*Attribute Name* i = *Attribute Name* j] [[**And** <Join Clause>] |⊥]
Group by Clause ::-	**Group by** [**Cube** | **Rollup**] <Attribute Clause>
Having Clause ::-	[*Alias* | *Aggregate Function Name* (*Attribute Name*)] <Comparison Operator> [*Attribute Value* | *Attribute Value List*]
Key:	The [and] brackets are delimitors. !<A>: A is required. *<A>: A is optional. <A ‖ B>: A or B. <A | B>: A exclusive or B. ⊥: empty clause. SQL langage elements are indicated in bold.

Figure 6: DWEB query model

5.2 Parameterization

DWEB's workload parameters help users tailoring the benchmark's load, which is also dependent from the warehouse schema, to their needs.

Just like DWEB's database paramameter set (Section 4.2), DWEB's workload parameter set (Table 3) has been designed with Gray's simplicity criterion in mind. These parameters determine how the query model from Figure 6 is instantiated. These parameters help defining the workload's size and complexity, by setting up the proportion of complex OLAP queries (i.e., the class of queries) in the workload , the number of aggregation operations, the presence of a Having clause in the query, or the number of subsequent drill down operations.

Here, we have only a limited number of high-level parameters (eight parameters, since $PROB\_EXTRACT$ and $PROB\_ROLLUP$ are derived from $PROB\_OLAP$ and $PROB\_CUBE$, respectively). Indeed, it cannot be envisaged to dive further into detail if the workload is as large as several hundred queries, which is quite typical.

Parameter name	**Meaning**	**Default value**
NB_Q	Approximate number of queries in the workload	100
AVG_NB_ATT	Average number of selected attributes in a query	5
AVG_NB_RESTR	Average number of restrictions in the query	3
$PROB\_OLAP$	Probability that the query type is OLAP	0.9
$PROB\_EXTRACT$	Probability that the query is an extraction query	$1 - PROB\_OLAP$
AVG_NB_AGGREG	Average number of aggregations in an OLAP query	3
$PROB\_CUBE$	Probability of an OLAP query to use the Cube operator	0.3
$PROB\_ROLLUP$	Probability of an OLAP query to use the Rollup operator	$1 - PROB\_CUBE$
$PROB\_HAVING$	Probability of an OLAP query to include an Having clause	0.2
AVG_NB_DD	Average number of drill downs after an OLAP query	3

Table 3: DWEB workload parameters

Remark: NB_Q is only an *approximate* number of queries because the number of drill down operations after an OLAP query may vary. Hence we can stop generating queries only when we actually have generated as many or more queries than NB_Q.

5.3 Generation algorithm

The pseudo-code of DWEB's workload generation algorithm is presented in Figures 7 and 8. The algorithm's purpose is to generate a set of SQL-99 queries that can be directly executed on the synthetic data warehouse defined in Section 4. It is subdivided into two steps:

1. generate an initial query that may either be an OLAP or an extraction (join) query;
2. if the initial query is an OLAP query, execute a certain number of drill down operations based on the first OLAP query. More precisely, each time a drill down is performed, an attribute from a lower level of dimension hierarchy is added to the attribute clause of the previous query.

Step 1 is further subdivided into three substeps:

1. the Select, From, and Where clauses of a query are generated simultaneously by randomly selecting a fact table and dimensions, including a hierarchy level within a given dimension hierarchy;
2. the Where clause is supplemented with additional conditions;
3. eventually, it is decided whether the query is an OLAP query or an extraction query. In the second case, the query is complete. In the first case, aggregate functions applied to measures of the fact table are added in the query, as well as a Group by clause that may include either the Cube or the Rollup operator. A Having clause may optionally be added in too. The aggregate function we apply on measures is always Sum since it is the most common aggregate in cubes. Furthermore, other

aggregate functions bear similar time complexities, so they would not bring in any more insight in a performance study.

We use three classes of functions and a procedure in this algorithm.

1. `Random_string()` and `Random_float()` are the same functions than those already described in Section 4.3. However, we introduce the possibility for `Random_float()` to use either a uniform or a gaussian random distribution. This depends on the function parameters: either a range of values (uniform) or an average value (gaussian). Finally, we introduce the `Random_int()` function that behaves just like `Random_float()` but returns integer values.

2. `Random_FT()` and `Random_dimension()` help selecting a fact table or a dimension describing a given fact table, respectively. They both use a gaussian random distribution, which introduces an access skew at the fact table and dimension levels. `Random_dimension()` is also already described in Section 4.3.

3. `Random_attribute()` and `Random_measure()` are very close in behaviour. They return an attribute or a measure, respectively, from a table intention or a list of attributes. They both use a gaussian random distribution.

4. `Gen_query()` is the procedure that actually generates the SQL-99 code of the workload queries, given all the parameters that are needed to instantiate our query model.

6 DWEB implementation

DWEB is implemented as a Java software. We selected the Java language to meet Gray's portability requirement. The current version of our prototype is able to generate star, snowflake, and constellation schemas, and suitable workloads for these schemas. Furthermore, since DWEB's parameters might sound abstract, our prototype provides an estimation of the data warehouse size in megabytes after they are set up and before the database is generated. Hence, users can adjust the parameters to better represent the kind of warehouse they need. Our prototype can be interfaced with most existing relational database management systems through JDBC. Database connexion, parameter selection, warehouse and workload generation, and workload execution are all accessible through a graphical interface.

Since we use a lot of random functions, we also plan to include in our prototype a better than standard pseudorandom number generator, such as the Lewis-Payne generator [LP73], which has a huge period, or the Mersenne Twister [MN98], which is currently one of the best pseudorandom number generators.

Finally, though our software is constantly evolving, its latest version is always freely available online [JG05].

7 Sample usage of DWEB

In order to illustrate one possible usage for DWEB, we tested the efficiency of bitmap join indices [OG95] on decision support queries under Oracle. Bitmap join indices are well suited to the data warehouse environment. They indeed improve the response time of such common operations as `And`, `Or`, `Not`, or `Count` that can operate on the bitmaps (and thus directly in memory) instead of the source data. Furthermore, joins are computed *a priori* when the indices are created. In such a context, we could compare the performances of various indexing techniques using one test warehouse and one or several workloads, or evaluate the efficiency of one or several given indexing techniques on various configurations

```
n = 0
While n < NB_Q do
    // Step 1:  Initial query
    // Step 1.2:  Select, From and Where clauses
    i = Random_FT() // Fact table selection
    attribute_list = ∅
    table_list = ft(i)
    condition_list = ∅
    For k = 1 to Random_int(AVG_NB_ATT) do
       j = Random_dimension(ft(i)) // Dimension selection
       l = Random_int(1, ft(i).dim(j).nb_levels)
       // Positioning on hierarchy level l
       hl = ft(i).dim(j) // Current hierarchy level
       m = 1 // Level counter
       fk = ft(i).intention.primary_key.element(j)
       // (This foreign key corresponds to ft(i).dim(j).primary_key)
       While m < l and hl.child ≠ NIL do
          // Build join
          table_list = table_list ∪ hl
          condition_list = condition_list ∪ (fk = hl.intention.primary_key)
          // Next level
          fk = hl.intention.foreign_key
          m = m + 1
          hl = hl.child
       End while
       attribute_list = attribute_list ∪ Random_attribute(hl.intention)
    End for
    // Step 1.2:  Supplement Where clause
    For k = 1 to Random_int(AVG_NB_RESTR) do
       condition_list = condition_list
          ∪ (Random_attribute(attribute_list) = Random_string())
    End for
    // Step 1.3:  OLAP or extraction query selection
    p1 = Random_float(0,1)
    If p1 ≤ PROB_OLAP then // OLAP query
       // Aggregate clause
       aggregate_list = ∅
       For k = 1 to Random_int(AVG_NB_AGGREG) do
          aggregate_list = aggregate_list ∪ Random_measure(ft(i).intention)
       End for
       // Group by clause
       group_by_list = attribute_list
       p2 = Random_float(0,1)
       If p2 ≤ PROB_CUBE then
          group_by_operator = CUBE
       Else
          group_by_operator = ROLLUP
       End if
       // Having clause
       p3 = Random_float(0,1)
       If p3 ≤ PROB_HAVING then
          having_clause = (Random_attribute(aggregate_list), ≥, Random_float())
       Else
          having_clause = ∅
       End if
../..
```

Figure 7: DWEB workload generation algorithm – Part 1

```
../..
    Else // Extraction query
       group_by_list = ∅
       group_by_operator = ∅
       having_clause = ∅
    End if
    // SQL query generation
    Gen_query(attribute_list, aggregate_list, table_list, condition_list,
       group_by_list, group_by_operator, having_clause)
    n = n + 1
    // Step 2:  Possible subsequent DRILL DOWN queries
    If p1 ≤ PROB_OLAP then
       k = 0
       While k < Random_int(AVG_NB_DD) and hl.parent ≠ NIL do
          k = k +1
          hl = hl.parent
          att = Random_attribute(hl.intention)
          attribute_list = attribute_list ∪ att
          group_by_list = group_by_list ∪ att
          Gen_query(attribute_list, aggregate_list, table_list, condition_list,
             group_by_list, group_by_operator, having_clause)
       End while
       n = n + k
    End if
End while
```

Figure 8: DWEB workload generation algorithm – Part 2

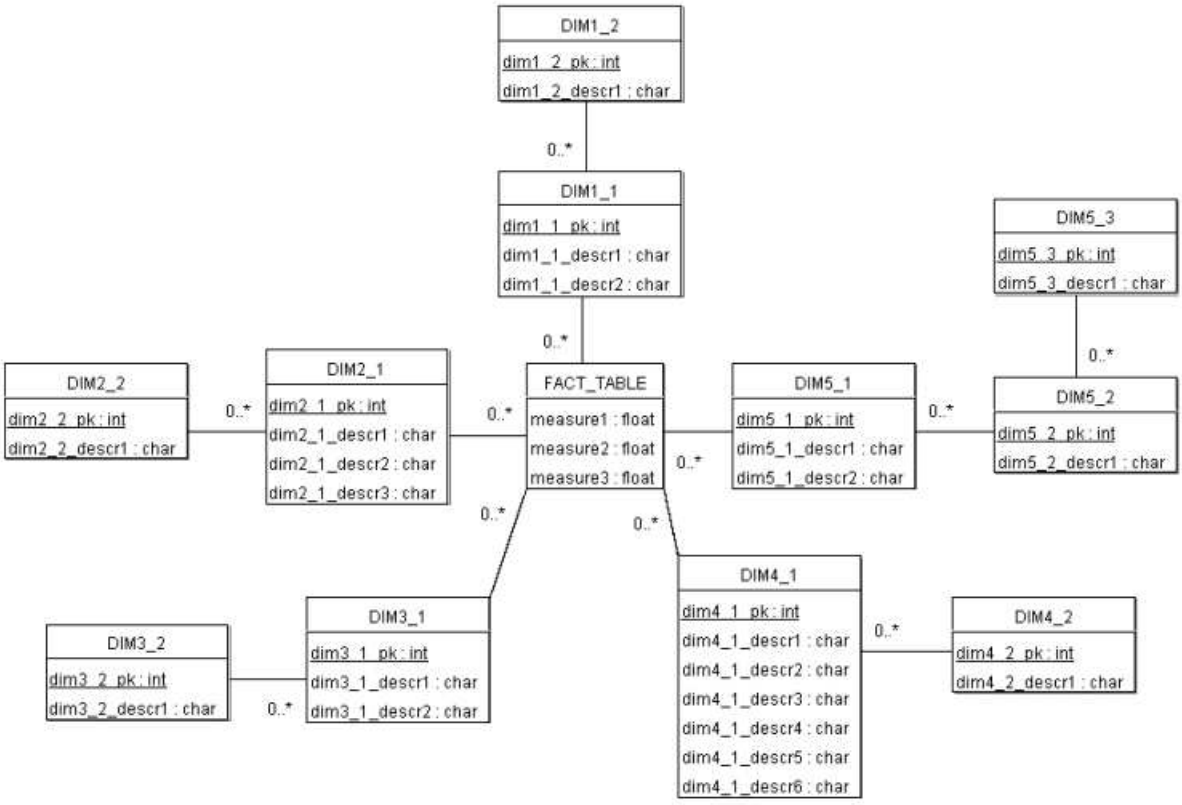

Figure 9: Sample snowflake schema generated by DWEB

of warehouses, etc. The aim of this particular example is to compare the execution time of a given workload on a given data warehouse, with and without using bitmap join indices.

First, we generated a data warehouse modeled as a snowflake schema whose conceptual schema is represented as a UML class diagram in Figure 9. The schema is organized around one fact table that is described by five dimensions (DIM1 to DIM5). Each dimension is hierarchical, DIM1 to DIM4 bearing two levels of hierarchy and DIM5 three levels. At the logical level, the fact table's primary key is consituted of the aggregation of the dimensions' lowest level primary keys (namely, $dim1\_1\_pk$ to $dim5\_1\_pk$). Hierarchy levels in dimensions are materialized by foreign keys, e.g., the primary key $dim5\_3\_pk$ of dimension hierarchy level DIM5_3 is a foreign key in DIM5_2. The fact table contains about 140,000 tuples, the dimension hierarchy levels about ten tuples on an average, for a global size of about 4 MB. Note that this is a voluntarily small example and not a full-scale test.

We applied different workloads on this data warehouse. $Workload$ #1 is a typical DWEB workload (see Section 5) constituted of fifty queries generated "by hand". 10% of these queries are extraction (join) queries and the rest are decision support queries involving OLAP operators (Cube and Rollup). In $Workload$ #1, we limited the queries to the dimensions' lowest hierarchy levels, i.e., to the star schema constituted of the fact table and hierarchy levels DIM1_1 to DIM5_1. $Workload$ #2 is similar to $Workload$ #1, but it is extended with drill down operations that scan the dimensions' full hierarchies (from the highest level to the lowest level). Thus, this workload exploits the whole snowflake schema.

To evaluate the efficiency of bitmap join indices, we timed the execution of these two workloads on our test data warehouse, first with no index, and then by forcing the use of five bitmap join indices defined on the five dimensions (for the lowest hierarchy levels in $Workload$ #1 and for the whole hierarchies in $Workload$ #2). To flaten any response time variation in these experiments, we replicated each test ten times and computed the average response times. We made sure *a posteriori* that the standard deviation was close to zero. These tests have been executed on a PC with a Celeron 900 processor, 128 MB of RAM, an IDE hard drive, and running Windows XP Professional and Oracle 9i.

Figure 10 represents the average response time achieved for $Workload$ #1 and #2, with and without bitmap join indices, respectively. It shows a gain in performance of 15% for $Workload$ #1, and 9.4% for $Workload$ #2. This decrease in efficiency was expected, since the drill down operations added in $Workload$ #2 are costly and need to access the data (bitmap join indices alone cannot answer such queries). However, the overall performance improvement we achieved was not as good as we expected.

We formulated the hypothesis that the extraction queries, which are costly joins and need to access the data too, were not fully benefiting from the bitmap join indices. To confirm this hypothesis, we generated two new workloads, $Workload$ #3 and #4. They are actually almost identical to $Workload$ #1 and $Workload$ #2, respectively, but do not include any extraction (join) queries. Then, we repeated our experiment following the same protocol.

Figure 11 represents the average response time achieved for $Workload$ #3 and #4, with and without bitmap join indices, respectively. This time, we obtained similar results than in our previous experiment (in trend): response time clearly increases when drill down operations are included into the workload. However, response time is now much better and the gain in performance is 30.9% for $Workload$ #3, and 19.2% for $Workload$ #4.

As a conclusion, we showed with this simple experiment how DWEB could be used to evaluate the performances of a given DBMS when executing decision support queries on a data warehouse. Of course, these experiments are not very significant *per se*, and do not do justice to Oracle, since we did not seek to achieve the best performance. We did not combine bitmap join indices with any other type of indices, neither did we use any knowledge about how Oracle exploits these indices, for instance. However, we illustrated how DWEB could be used for performance evaluation purposes. These experiments could also

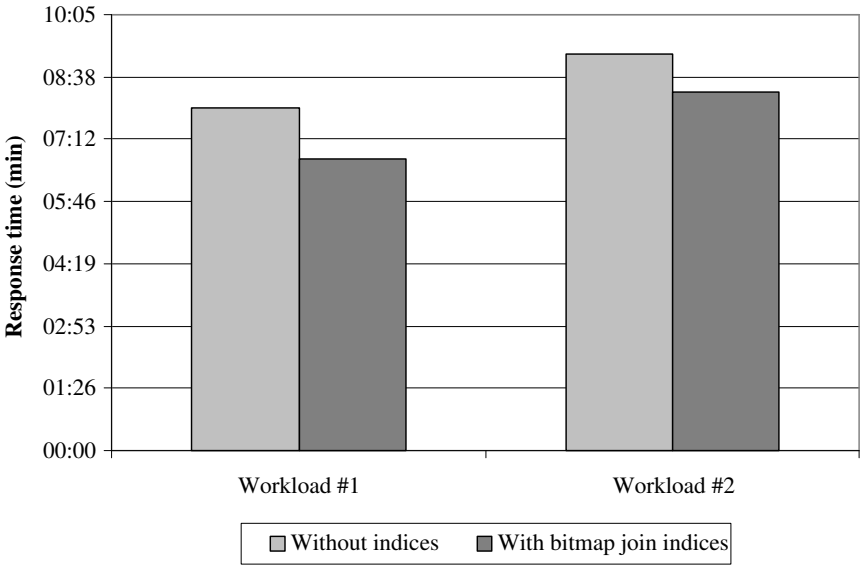

Figure 10: Workload #1 and #2 test results

be seen as a (very basic) performance comparison between two different data warehouse architectures (star schema and snowflake schema). Our results indeed conform to the well-known fact that introducing hierarchies into a star schema induces more join operations in the decision support queries, and hence degrade their response time. Finally, we were also able to witness the impact of costly join operations on a data warehouse structure that is not properly indexed to answer such queries. This might lead us to further increase the default probability of running OLAP queries in our benchmark (see Section 5.2).

8 Conclusion and perspectives

We aimed in this paper at helping data warehouse designers to choose between alternate warehouse architectures and performance optimization techniques. For this sake, we proposed a performance evaluation tool, namely a benchmark called DWEB (the *Data Warehouse Engineering Benchmark*), which allows users to compare these alternatives. To the best of our knowledge, DWEB is currently the only operational data warehouse benchmark. Its main feature is that it can generate various ad-hoc synthetic data warehouses and their associated workloads. Popular data warehouse schemas, such as star schemas, snowflake schemas, and constellation schemas can indeed be achieved. We mainly view DWEB as an engineering benchmark designed for data warehouse and system designers, but it can also be used for sheer performance evaluations. Note that the database schema of TPC-DS, the future standard data warehouse benchmark currently developped by the TPC, can be modeled with DWEB. In addition, though DWEB's workload is not currently as elaborate as TPC-DS's, it is also much easier to implement. It will be important to fully include the ETL process into our workload, though, and the specifications of TPCD-DS and some other existing studies [LR98] might help us. Finally, we tried to provide in this paper the most comprehensive specifications for DWEB, so that our benchmark can be implemented easily by other data warehouse designers and researchers. We also illustrate with a practical case how this tool can be used.

When designing DWEB, we tried to grant it the characteristics that make up a "good" benchmark according to Gray: relevance, portability, scalability, and simplicity. To make DWEB relevant for evaluating the performance of data warehouses in an engineering context, we designed it to generate different data warehouse schemas and workloads. However, we now need to further test DWEB's relevance on

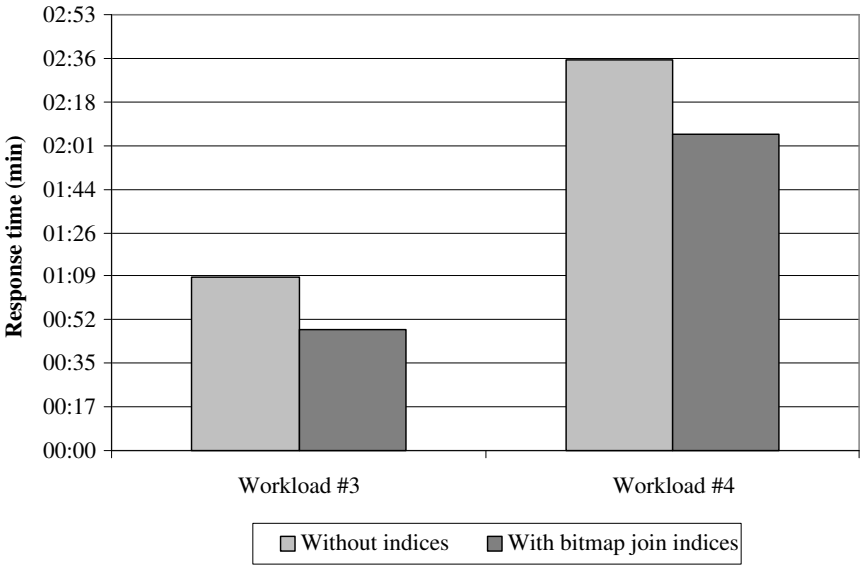

Figure 11: Workload #3 and #4 test results

real cases. To achieve this goal, we plan to compare the efficiency of various index and materialized view selection techniques (including some of our own proposals, which was a motivation for designing DWEB in the first place). We also made DWEB very tuneable to reach both the relevance and scalability objectives. However, too many parameters make the benchmark complex to use and contradict the simplicity requirement. Though it is impossible to achieve both a high simplicity and a high relevance and scalability, we introduced a layer of high-level parameters that are both simpler than the potentially numerous low-level parameters, and in reduced and constant number. DWEB might not be qualified as a simple benchmark, but our objective was to keep its complexity as low as possible. Finally, portability was achieved through our Java/JDBC implementation.

This preliminary work opens up many perspectives for developing and enhancing DWEB. In this paper, we assumed an execution protocol and performance metrics were easy to define for DWEB and focused on the benchmark's database and workload model. A more elaborate reflexion about these topics might be fruitful, especially since two executions of DWEB using the same parameters produce different data warehouses and workloads. This is interesting when, for instance, one optimization technique needs to be tested against many databases. However, it must also be possible to save a given warehouse and its associated workload to successively test different optimization techniques on it.

To work toward the simplicity objective, we also plan to work at making DWEB's schema more understandable, for example by defining domain-specific data dictionaries so that meaningful values are associated to table names, attribute names and values (e.g., Sales and Region instead of FACT_TABLE and DIM_1).

We are also currently working on warehousing complex, non-standard data (such as multimedia, multistructure, multisource, multimodal, and/or multiversion data). Such data may be stored as XML documents. Thus, we also plan a "complex data" extension of DWEB that would take into account the advances in XML warehousing.

Finally, more experiments with DWEB should also help us acquire experience on using the benchmark and maybe propose sounder default parameter values. We also encourage other data warehouse designers and researchers to report on their own experiments with DWEB.

References

[Bal93] Carrie Ballinger. *TPC-D: Benchmarking for Decision Support*. The Benchmark Handbook for Database and Transaction Processing Systems. Morgan Kaufmann, 1993.

[Bha96] Ramesh Bhashyam. TCP-D: The Challenges, Issues and Results. *SIGMOD Record*, 25(4):89–93, December 1996.

[BMC00] BMC Software. Performance Management of a Data Warehouse. http://www.bmc.com, 2000.

[Dem95] Marc Demarest. A Data Warehouse Evaluation Model. *Oracle Technical Journal*, 1(1):29, October 1995.

[DS00] J Darmont and M Schneider. Benchmarking OODBs with a Generic Tool. *Journal of Database Management*, 11(3):16–27, Jul-Sept 2000.

[Gra93] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, second edition, 1993.

[Gre04a] Larry Greenfield. Performing Data Warehouse Software Evaluations. http://www.dwinfocenter.org/evals.html, 2004.

[Gre04b] Larry Greenfield. What to Learn About in Order to Speed Up Data Warehouse Querying. http://www.dwinfocenter.org/fstquery.html, 2004.

[Inm02] W.H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, third edition, 2002.

[JG05] B. Joubert and S. Guesmoa. DWEB Java prototype v0.3. http://bdd.univ-lyon2.fr/download/dweb.tgz, 2005.

[KR02] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, second edition, 2002.

[LP73] T.G. Lewis and W.H. Payne. Generalized feedback shift register pseudorandom number algorithm. *ACM Journal*, 20(3):458–468, 1973.

[LR98] Alexandros Labrinidis and Nick Roussopoulos. A performance evaluation of online warehouse update algorithms. Technical Report CS-TR-3954, Deptartment of Computer Science, University of Maryland, November 1998.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *CM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[Obj03] Object Management Group. *Common Warehouse Metamodel (CWM) Specification version 1.1*, March 2003.

[OG95] P.E. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[PCTM03] John Poole, Dan Chang, Douglas Tolbert, and David Mellor. *Common Warehouse Metamodel Developer's Guide*. John Wiley & Sons, 2003.

[Pen03] Nigel Pendse. The OLAP Report: How not to buy an OLAP product. http://www.olapreport.com/How_not_to_buy.htm, December 2003.

[PF00] Meikel Poess and Chris Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Record*, 29(4):64–71, December 2000.

[PSKL02] Meikel Poess, Bryan Smith, Lubor Kollar, and Per-Ake Larson. TPC-DS: Taking Decision Support Benchmarking to the Next Level. In *ACM SIGMOD 2002, Madison, USA*, June 2002.

[Tra98] Transaction Processing Performance Council. *TPC Benchmark D Standard Specification version 2.1*, February 1998.

[Tra03a] Transaction Processing Performance Council. *TPC Benchmark H Standard Specification version 2.1.0*, August 2003.

[Tra03b] Transaction Processing Performance Council. *TPC Benchmark R Standard Specification version 2.1.0*, August 2003.

[Tra04] Transaction Processing Performance Council. Web site. http://www.tpc.org, 2004.