Dealing with Slow-Evolving Fact: A Case Study on Inventory Data Warehousing [*†]

Chung-Min Chen Munir Cochinwala Elsa Yueh
Telcordia Technologies (formerly Bellcore)
Morristown, NJ 07960
chungmin@research.telcordia.com

Abstract

Data Warehousing for INventory management (DWIN) is a production project at Telcordia aimed at providing telecommunications service providers with decision support functions for inventory control and monitoring. In this paper, we report some interesting issues related to the design of the data warehouse. Specifically, we will discuss the issues of slow-evolving fact, transaction-oriented fact table, and large dimensions. We also propose the concept of *virtual data cubes* and show its usefulness. We address these issues through a data mart case study and present benchmarking results. Finally, based on the experiences learned, we discuss potential research issues that may benefit the data warehousing and OLAP practice.

1 Introduction

To help ensure a fast response to customer demands for repair and new service, large telecommunications companies usually maintain, in its region of operation, multiple field locations (or warehouses) that stock and distribute equipments. Typical equipments are *plug-ins*, which are modular printed circuit boards that provide switching, transmission, and other network functions, and their parts. When a central office calls for a spare plug-in(s), the warehouse dispatches the requested equipment(s) to the destination. The warehouses also stock plug-ins that are called back for repair or replacement.

Telcordia's SIMM (Stock Inventory Management Module) and PICS (Plug-in Inventory Control System) products [1] help companies control and track this day-to-day process [2]. The PICS is a 25-year old mainframe-based application; the SIMM is a client-server application based on relational database technologies. Both systems have been deployed at several regional telephone companies. Together, they control over $70 billions of equipment investment.

A typical regional service provider keeps *spare* plug-in inventory worth $350 millions to $500 millions. Because accurately tracking these spares requires a complex series of tasks, it is not unusual for 20% of a company's inventory assets to be sitting idle in a warehouse. As much as $50 millions to $100 millions worth of plug-ins may be wasting warehouse space because they are damaged, obsolete, or simply forgotten excess. No company can afford to carry that kind of idle inventory. In today's competitive marketplace, companies must cut costs to remain competitive. Cutting inventory costs requires thorough analysis on historical data for performance/effectiveness evaluation and demand forecasting.

Telcordia's Data Warehousing for INventory management (DWIN), created out of this demand, is part of Telcordia's endeavor to provide an integrated suite of decision support tools to telephony service providers. The DWIN, offered as an add-on to SIMM (or PICS), is a suite of customized, plug-and-play data warehousing applications that features broad decision support functions for inventory management. The system gives managerial personnel a global picture on the effectiveness/performance of the company's day-to-day inventory practice by providing multidimensional views on historical data. The system also enables the executives to make better decisions by allowing mining and drilling to detail data.

In this paper, we report some interesting design issues encountered in the DWIN project. We present

* © 1999 Telcordia Technologies, Inc. All rights reserved.
† Other contributors include Louis Defosse, Dennis Swartz, Christin Rastguelenian and Ramkrushna Pandya.

[1] SIMM and PICS are registered trade marks of Telcordia Technologies.

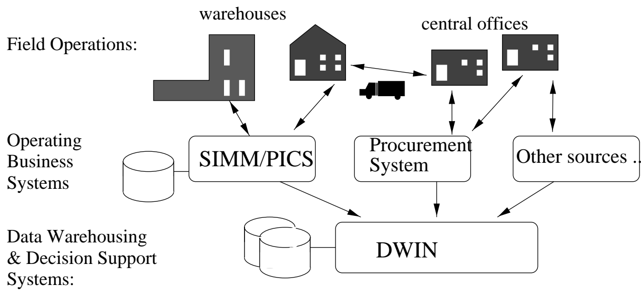


Figure 1: Overview of SIMM/DWIN Environment

these issues through a case study on a data mart that tracks the stock level of equipments in repair. We show and argue that, due to the nature and size of the data in our practice, the traditional design concept of recording "measurable fact" in a star schema is inappropriate and inefficient. Specifically, the sources of the problem are *slow-evolving fact* (data whose values change less frequently over time) and *large dimensions*. We then present a solution based on the concept of "*transaction-oriented*" fact table and the "*virtual cube*" approach. A transaction-oriented fact table contains transaction-detail records rather than "measures". A virtual cube is a data cube defined on some other data cube(s) and whose values are to be materialized on-the-fly. We show, through benchmarking, that this design is more efficient than the traditional design in terms of space requirement, maintenance cost, and query performance. Finally, we discuss potential research issues that arise from this case study.

2 Project Description

Figure 1 provides an overview of the SIMM/DWIN data warehousing environment. The SIMM processes and records every inventory transactions. The transactions are kept in the database for a certain period of time before they are permanently discarded. The DWIN extracts data, periodically, from SIMM and other sources. The extracted data are cleansed, integrated, and loaded into the data warehouse to facilitate multidimensional analysis and ad hoc queries. In this paper, we will not elaborate further on the components and specific implementation of DWIN as they resemble the typical data warehousing approach [9]. Rather, we will discuss design issues that would affect the performance of the data warehouse and, in our opinion, whose ultimate solutions require further research.

To begin with, we give a broad description of the environment in which the SIMM (or PICS) is operated. Consider the inventory practice of an operating telephone company. The company stockpiles a large number of *equipments* at many *field locations*. An equipment can be in one of many states: *spare*, *awaiting repair*, *in-repair*, etc. To simplify discussion, we will focus on equipments in-repair in this paper. Each day, the workers at the field locations pack the equipments that need to be repaired into packages. Equipments of the same kind are packed into a single package, called a *repair order*, which is then *shipped* to the manufacturer or contractor for repair. When the repair is done, the package is *returned* to the same location that shipped the package.

The SIMM records the shipping and returning dates of each repair order. These are kept in a table called `Work` that records every transaction (not necessarily a repair transaction) at every location. The `Work` table, whose partial schema is shown below, has more than 40 attributes, including `trans_type` (whose values are "repair", "replace" "transfer", "dispose", etc), `init_date`, `complete_date`, `equipment`, `location`, `qty` and other related information such as customer and shipping information.

```
Work (trans_type, init_date, complete_date,
      equipment, location, qty, ...)
```

When a record's value on `trans_type` is "repair" it represents a repair order, and its values in `init_date` and `complete_date` indicates the shipping and returning dates, respectively. A `NULL` value in `complete_date` indicates that the order has not been returned. We will assume the predicate `NULL > date`, where `date` is any legitimate date value, always yields `TRUE`.

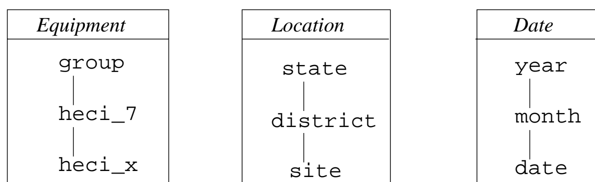
Periodically, stale records are purged from the `Work` table to make room for new records and to maintain fast transaction response time (no longer than a few seconds). The purge is usually performed at a frequency of 3 to 9 months, depending on the actual load (typically, there are between 20,000-50,000 transactions per day, where each transaction may create or update a number of `Work` records). Yet, the `Work` table may contain millions of records at any instance of time.

The DWIN data warehouse, which extracts and loads historical data (periodically) from the SIMM database, is created as a platform to facilitate OLAP and complex ad hoc queries. In this paper we will concentrate on tracking repair activities. Two example queries are:

Q1. *Show me the number (or value) of units in repair by any combination of equipment type, date, and location.*

Q2. *Among the repairs issued in the last 6 months, show me the number of repair orders and units that are completed within n days ($n = 1, 2, \ldots$).*

Our task, then, is to come up with an effective data mart design that would answer the above kinds of queries efficiently. In our development, we have adopted a commercial Relational OLAP (ROLAP) product, which includes a DBMS and an add-on data mart development



dimension tables: Equip (equip_key, group, heci_7, heci_x, ...)
Loc (loc_key, state, district, site, ...)
Date (date_key, year, month, date)

fact table: RepairStock (equip_key, loc_key, date_key, stk, ...)

Figure 2: Dimension Hierarchies and Schema

tool. Since we are using a commercial-off-the-shelf product, all our discussion that follows is based on the premise that one can not change the kernel of the underlying DBMS. Any design philosophy or technique used must be based on something supported by the DBMS. We have surveyed most major commercial ROLAP products and found that they provide similar support with respect to our requirements. Thus, any discussion or claim in this paper should not be interpreted as specific to the DBMS we used. We will discuss possible solutions from the perspective of DBMS in Section 6.

3 First Attack: Star Schema with Measurable Fact

Figure 2 shows the dimension hierarchies and schema of a data cube that is aimed at answering Q1. We will refer to this schema design as A and use the notation `A.x` to refer to a table named `x` in schema A. The design follows the normal "star schema" design convention [3, 9, 6]. The *Equipment* dimension is a catalog of all types of equipments and parts used by the telephone company. This dimension has three level and is based on a widely adopted classification scheme for switch equipments. Each member (value) in the `group`, `heci_7`, or `heci_x` level represents a different sub-class of equipments or parts. The hierarchy in the `Location` dimension is divided into state, district, and site. Each site is a (mini-)warehouse or storage that stocks and distributes equipments. Three dimension tables: `Equip` (for equipments), `Loc` (for locations) and `Date` are created. For concise purpose, we have omitted other description attributes, the so called *properties*, associated with the dimension tables. For example, the actual `Loc` table in DWIN contains other attributes such as address, capacity, ans so on.

The fact table `RepairStock` contains a "*measure*" attribute `stk`, which represents the number of units that are in repair of a specific equipment, at a specific location, on a specific date. In addition, it has a foreign key field that references the primary key of each dimension table.

We soon learned that this design was very inefficient in light of the characteristics of our field data. First, the equipment dimension is large: a typical regional telephone company may have between 90,000-200,000 different kinds of equipments (at the `heci_x` level) ! And some of our customers maintain up to 4,000 sites spanning all the states they operate.

If on an average day each site has 500 different equipments under repair, then over a period of 12 months, the fact table `RepairStock` would contain $4,000 \times 500 \times 365 \approx 730$ million records. Note that we do not store fact records with zero `stk` value. If a combination of `Equip.equip_key`, `Loc.loc_key`, and `Date.date_key` does not find a match in the fact table, then it is interpreted that the `stk` value corresponds to this combination is zero. This approach avoids the sparse cube problem.

The actual fact table, which includes other measure attributes (such as cost of repair) and foreign keys for additional dimensions (such as manufacturers and shipping vehicles/methods), amounts to 22 GB for one-year worth of data. A single-purpose data mart with a 22 GB fact table certainly is not very attractive, as it not only consumes a lot of storage space, but also affects the query performance.

This design has another problem: it is inept to answer the kind of queries like Q2. To answer Q2 (or any other query related to repair), another data cube must be created, which will in turn take up more system resources, as well as increasing the ETL (extract, transform and load) overhead.

4 The Solution: Virtual Cubes on Transaction-Oriented Fact Table

The size of the fact table in the previous design has been a serious concern to our customers. Fortunately, we found that for most of the equipments, the repair stock at a location tends to remain unchanged over a period of time, typically ranging from a few days to a few weeks. In other words, during that period, no new repair orders of that specific equipment type are shipped out and nor have repaired ones returned. This property, which we termed "slow-evolving fact", suggests a possible remedy to the large fact table problem. To see this, suppose that equipment e has q units under repair at site s on date d, and that for the following 10 days there are no activities on equipment q. With the previous design, the fact table `A.RepairStock` will store 10 redundant records $[e, s, D, q]$, where $D = d+1, d+2, \ldots, d+10$, that convey basically the same information (that the repair stock is q). These redundant records cannot be omitted, because otherwise it would be interpreted to have a zero `stk` during the ten-day period.

While most OLAP tools provide solutions for the

sparse cube problem, they do not handle slow-evolving fact well. As a work around to avoid the redundancy problem of slow-evolving fact, we came up with an alternative design. We will refer to this new schema design as B. The first goal of the new design is to reduce the redundancy in the fact table. To achieve this at the schematic level, we extract, for each repair order in table SIMM.Work, the shipping and returning dates and store them in the following fact table:

```
RepairOrder (equip_key, loc_key, out_date,
             in_date, qty)
```

Each record in RepairOrder represents a repair order: equip_key indicates the equipment enclosed in the package and is a foreign key to Equip; loc_key indicates the location that ships the order and is a foreign key to Loc; out_date and in_date are the shipping and returning dates, respectively, and are foreign keys to Date. Attribute qty is the number of units enclosed in the order.

It is worth noting that, unlike a conventional fact table that records "incidental' fact (such as point sales), the RepairOrder fact table contains (extracted) transaction information. We called it a "transaction-oriented" fact table. This concept is coincided in [9] by the term "transaction-level table" which, unlike our case, contains only incidental quantities and does not deal with slow-evolving data. By including a time interval component in each record, the fact table B.RepairOrder eliminates the redundant records in A.RepairStock. It is not hard to see that if the average interval to the next shipping/returning of the same equipment type at a location is N days, then B.RepairOrder stores only $1/N$ records of A.RepairStock.

Note that the choice between approaches A and B really depends on the value of N. Since B.RepairOrder has a larger record size than A.RepairStock and requires extra cost to answer queries, B may not be advantageous if N is close to 1 (e.g. 1.2).

4.1 Virtual Cubes

Now we have to find a way to derive the repair stock (dimensioned by equipment, location, and date) from the RepairOrder table. We do not want to materialize and store the entire data cube because then we are going back to where we started (namely requiring large space with mostly redundant records). Also, from a practical point of view, a full materialization is unnecessary as the size of the equipment dimension (which contains 200,000 members) would have prohibited a user from visualizing the entire cube at a time.

Observing these, we have built a customized program that takes dimension parameters as input from the user (e.g. by specifying a sub-class of equipments to aggregate) and dynamically generates a partial cube or cross-tab. The program works in an interactive mode and enables the user to slice-and-dice and drill up and down on the "virtual cube". We will not detail the program here as it uses vendor-specific report generating tools and is too detailed for our current discussion. However, the idea of *virtual cube* can be equivalently expressed in an SQL-like language as shown below.

```
create virtual cube RepairStockVC (alias c) as
dimension Equip, Location, Date
virtual fact stk = f( );

define function f ( ) as
select  SUM(r.qty)
from    RepairOrder (alias r)
where   r.equip_key = c.equip_key and
        r.loc_key = c.loc_key and
        r.out_date ≤ c.date_key < r.in_date
```

The keyword virtual indicates that the values of the cube or fact is to be evaluated on-the-fly. The dimension clause specifies three pre-existing dimension tables Date, Equip, and Location (whose schemas are given in Figure 2). The virtual fact clause specifies an attribute stk whose value is computed through function f. Given a point (c.equip_key, c.loc_key, c.date_key) in the virtual cube, where c is an alias of RepariStockVC, function f selects from RepairOrder those records that match c.equip_key and c.loc_key and whose repair interval [out_date, in_date] covers c.date_key. It then performs a SUM aggregate on the repair quantity of the selected records.

Technically speaking, one may achieve the same effect as the above by defining a view in SQL:

```
create view RepairStockView as
select  r.loc_key, r.equip_key, d.date_key,
        SUM(r.qty)
from    RepairOrder r, Date d
where   r.out_date ≤ d.date_key < r.in_date
group by r.loc_key, r.equip_key, d.date_key
```

Though this statement is shorter in length, we are still in favor of the virtual cube expression, for the following reasons: (1) the virtual cube expression, with explicit dimension and fact clauses, is more intuitive and self-contained (note with the view expression, one must explicitly relate the view to the dimension tables at the meta-data level), and (2) the virtual cube expression is more amenable to optimization, as its structure and semantics are better understood than the view expression, which does not distinguish between dimensions and fact.

The transaction-oriented fact table RepairOrder also enables us to answer query Q2 in a virtual cube, as

shown below. The RepairDays dimension is a pre-existing table that holds a single attribute num_repair_days (with a bounded integer domain).

```
create virtual cube RepairTimeVC (alias c) as
dimension Equip, Location, RepairDays
virtual fact (num_repair_orders,
              num_repair_units) = f( )


define function f ( ) as
select  count(*), SUM(r.qty)
from    RepairOrder (alias r)
where r.equip_key = c.equip_key and
   r.loc_key = c.loc_key and
   today_date - r.out_date ≤ 180 and
   r.in_date - r.out_date = c.num_repair_days
```

5 Benchmarking

In this section, we compare and provide benchmark results for the two different designs. Due to resource restriction and sensitivity of the real data, we have created a synthetic database for benchmarking purpose. The database is created on a Sun SparcStation-20, with dual processors and 512 MB memory, that runs the same DBMS we use in the production. The benchmark database reflects the characteristics of the real data but is smaller in size. We assume a workload of 20,000 repair orders per day over a period of 150 days. Each order is completed within 20-30 days. Based on these numbers, a Work table is created that contains $20,000 \times 150 = 3M$ (million) records. Tables A.RepairStock and B.RepairOrder are derived accordingly from Work. All tables are properly indexed to facilitate queries. We assume the value distributions of all attributes are uniform and mutually independent. The following table shows other statistics of the tables. The CurRepInv table is an on-line SIMM table that keeps track of the *current* repair inventory level (more details later).

	# of records	record size (byte)	table size
SIMM.Work	3M	424	1.27 GB
SIMM.CurRepInv	5M	40	200 MB
A.RepairStock	75M	16	1.2 GB
B.RepairOrder	3M	20	60 MB

Since the average order completion time is 25 days, B.RepairOrder contains only 1/25 the number of records of A.RepairStock (as the former avoids storing repetitive stock values). Although B.RepairOrder requires one extra attribute, its total size is still much less than that of A.RepairStock. This economic use of space by design B also translates into saving in query running time as fewer disk accesses are needed.

During the benchmarking, all gauged experiments were run under a standalone mode without interference from other concurrent users/processes. We flushed out the DBMS buffers before each run so the gauged statistics can be compared on a fair baseline. We have also tuned the DBMS's query optimizer, for example, by giving hints and re-ordering query predicates, to ensure that it chooses a best possible plan.

Maintenance Costs

The maintenance costs include data mart creation and update. Both designs have the same maintenance cost on dimension tables: the Loc and Equip tables are created by extracting from SIMM and other sources; the Date table is generated automatically by the vendor tool. The procedures to create/update the fact table are quite different between the two designs. For design B, the procedure is a simple selection query (with projection) from the Work table. To do this incrementally, we keep a variable $LAST_UPDATE_DATE, which records the date that the RepairOrder table was last updated. To refresh, we extract a subset of Work defined by the following query:

```
select   equipment, location, init_date,
         complete_date, qty
from     Work
where    trans_type = 'repair' and
         (init_date > $LAST_UPDATE_DATE or
         (complete_date ≠ NULL and
          complete_date > $LAST_UPDATE_DATE))
```

The second predicate (init_date > ...) qualifies newly shipped repair orders that must be inserted into RepairOrder. The last predicate (complete_date > ...) indicates the receipt of a returned order whose corresponding record in RepairOrder must be updated. The update can be performed at arbitrary frequencies (e.g. daily or weekly) so long as every update to Work is captured before it is permanently purged by SIMM.

For design A, there are two possible ways to update the fact table RepairStock, but neither is perfect. The first approach relies on the SIMM table CurRepInv(loc, equip, current_units_in_repair,...), which records, for each equipment and location, the *current* number of units in repair. While it is straightforward to update RepairStock by selecting directly from SIMM.CurRepInv, it has some problems. First, it is not possible to create/recover an instance of RepairStock, which traces the repair stock information back to a previous date, based solely on the content of CurRepInv (because it contains only current information). Secondly, to keep historical data, RepairStock must be updated on a daily basis, as the current values in CurRepInv may be overwritten the next day. These restrictions result in little flexibility in update schedules. We refer to this approach as A1.

Another alternative to update A.RepairStock is to compute it from table SIMM.Work. Logically speaking,

	daily	weekly
A1	31 s.	$7 \times 31s.$
A2	743 s.	7,118 s.
B	26 s.	292 s.

Table 1: Maintenance Cost

	q1	q2	q3
result size	28,000	831+7	1
A	1,927 s.	2.02+0.20 s.	N/A
B	455 s.	0.96+0.23 s.	0.28 s.

Table 2: Query Performance

this procedure is equivalent to the SQL join query that defines `RepairStockView` (see Section 4), except that we are now selecting from `SIMM.Work` rather than from `B.RepairOrder`. However, unlike the materialization of `B.RepairStockView`, the procedure to create/update `A.RepairStock` directly from `Work` is much more time-consuming and cumbersome, because it requires a join between `Date` and the large on-line `Work` table. We refer to this approach as A2.

Table 1 shows the update cost of the different approaches from our benchmark. The cost, measured in terms of actual elapsed time (in seconds), reflects the overhead of extracting necessary incremental data from the on-line tables (`Work` or `CurRepInv`) and updating the fact table (`A.RepairStock` or `B.RepairOrder`). The corresponding SQL queries can be found in Appendix. This statistic is obtained using a trace facility of the DBMS. We do not report number of physical disk accesses from the trace facility as some of the numbers are not consistent and we suspect their reliability. We measured the costs for daily and weekly updates. For A1, however, the weekly cost represents a 7-day accumulation, as A1 is not eligible for weekly update.

Among the three, A2 has the worst maintenance cost, which is at least one order of magnitude lager than the other two, because it involves a group-by clause and requires either a sorting or hashing on the `Work` table. A1 and B, whose extraction procedures are simple selection queries (without group-by), have similar cost. Though A1 selects from a smaller table (`SIMM.CurRepInv`), it produces a larger output (500K records per day, containing every combination of location and equipment that has a non-zero repair stock) than B does (which selects from `SIMM.Work` but produces only 40K records per day—20K new orders and 20K completed orders).

Query Performance

To evaluate queries, B requires extra CPU time to (partially) materialize the virtual cube, whereas A has the answer readily available in the materialized fact table. On the other hand, B may require less I/O accesses because its fact table `RepairOrder` is only 1/25 the size of A's fact table `RepairStock`. As an attempt to evaluate their performance, we ran and gauged the following representative queries against both designs:

q1 : Show, for each day in the last 7 days, the repair stock at each location.

q2 :(1) First, show location X's daily repair stock by equipment for last week, then (2) show the weekly repair stock by equipment.

q3 : Show, among the repair orders issued on a specific `heci_7` equipment class during the last 80 days, the number of orders that are completed within 25 days.

We translated the above queries into corresponding SQL queries for designs A and B (the SQL queries for q1 are given in Appendix; those for q2 and q3 can be formed similarly and are omitted). For design A, queries q1 and q2 select from `RepairStock`; for design B, q1 and q2 select from `RepairStockView`. Query q3 is issued only for B against a virtual view equivalent to `B.RepairTimeVC` (see definition in Section 4).

It is not hard to see that q1 is a combination of slice (on the date dimension) and roll-up (on the equipment dimension). Query q2 slices on two dimensions (location and date), followed by a roll-up (on the date dimension). Query q3 contains a slice operation on each of the three dimension of the virtual cube `RepairTimeVC`.

Table 2 shows the result size (# of records) and actual elapsed time of the queries. To our surprise, B outperforms A in all cases. The difference is most significant for q1, which produces the largest result set. We found that B's better performance is largely due to the compact size of its fact table `RepairOrder`, which requires less, and mostly clustered, disk accesses to evaluate the queries. Though B incurs some materialization overhead (with group by clause), the overhead is insignificant when compared to A's excessive I/O cost to access the large, fully materialized cube `A.RepairStock`.

Extensibility

One important principle in data warehouse practice is that the design must be as general as possible to answer a broad range of ad hoc queries, some of which may not even be foreseen by the users at the beginning of the project [1]. Following the principle prevents the creation of a large number of small, narrow-purposed data marts (the so called "mushrooms" approach), which may turn the maintenance job into a nightmare. From this perspective, design B is apparently in favor. Virtually, any query related to equipments in-repair can be answered on-the-fly by deriving from the physical data cube `RepairOrder`, because it contains the details for every repair transaction. Design A, in contrast, is less extensible. For example, to answer Q2, another data

cube must be created, which incurs additional maintenance cost.

6 Concluding Remarks and Research Issues

We have described our design for a data mart whose goal is to provide long-term (as opposed to one-time shot) decision support functions, including multidimensional analysis as well as ad hoc queries, for inventory management. In the following list, we summarize the notable issues we encountered and suggest possible OLAP research problems.

- *Transaction-oriented fact table*: We have shown how a transaction-oriented fact table can efficiently support OLAP queries involving slow-evolving, status (as opposed to incidental) fact. The transaction-oriented fact table, upon which other (virtual) data cubes can be built, reduces the space and maintenance overhead, as well as providing better query performance and extensibility.

  This approach, however, does not always lead to advantages. When the fact is "fast-evolving", the transaction-oriented fact table may contain, in the worst case, as many records as a regular fact table. In such a case, the advantages in space saving, maintenance overhead and query performance vanish. It is desirable to set up and quantify criteria when using a transaction-oriented fact table will be advantageous.

- *Slow-evolving fact*: Slow-evolving fact is common in inventory applications where stock levels (not necessarily repair stock) are rather stable. Other examples include measures in science disciplines (for example, the soil pH level sampled at a geographical area may not change over a long period). The current OLAP tools are adequate for "incidental' fact (e.g. daily sales), but do not handle slow-evolving data well—usually they have to repeat the same value in a cluster of points in the multidimensional space. (Note that the problem is different from the classic sparse cube problem [11]).

  We feel a strong need for DBMS/OLAP vendors to support slow-evolving in their products (perhaps through some "version" or temporal mechanism).

- *Virtual cubes*: Virtual cubes are data cubes whose values are materialized on-the-fly, perhaps partially, from other cubes as the end-user slice-and-dice interactively. It is especially useful for very large data cubes that prohibit either a full materialization (due to space and computing resource constraints) or a full visualization (due to human-computer interface limitation).

  Implementation of virtual cubes generates some interesting research issues. One is the inclusion of virtual cubes as a first-class database object, rather than just a flat view, into the DBMS. The word "virtual cube" appears in a recent document that proposes a standard interface to access multidimensional sources [5]. The report details on the interface but does not formalize virtual cubes (neither syntactically nor semantically).

  The other issue is query optimization. As a data cube is materialized dynamically and partially, the best execution plan may change according to the input parameters (e.g. drill-down level and slice ranges). For instance, to evaluate the virtual cube `B.RepairStockVC`, one possible plan is to iterate through the key values in the dimension tables, and evaluate function `f()` against the physical table `B.RepairOrder`. This plan would be very inefficient because most of the iterations would result in a zero or repeated value for the aggregate `SUM(r.qty)`. A possibly better way is to scan `B.RepairOrder`, compute only non-zero aggregates, and hash them to the corresponding point in the multidimensional grid.

  The work on interactive aggregates [7] may shed some light on discovering efficient optimization techniques for virtual cubes. Caching and re-using materialized query results [10, 4] may also help on improving query performance of virtual cubes. Finally, efficient algorithms are needed to pre-compute a selective subset of aggregates in the cube lattice.

- *Large dimensions*: De-normalized star schemas in a ROLAP model usually cause data redundancy in the dimension tables, because the value of a non-leaf member v must appear in each record that is a leaf descendant of v. Such data redundancy is usually tolerated in exchange for better performance, as it avoids the expensive joins that would have been required if the dimension table(s) are normalized (in a "snow-flake" fashion). For large dimensions, however, the redundancy causes very poor space utilization and non-clustered I/O accesses.

  In our case, the `Equip` dimension table may have as much as 37% of redundant data. We were reluctant to normalize `Equip` due to the performance concern (frequent large joins). We feel this problem is better addressed from the DBMS, which should provide an efficient implementation of large dimensions, yet allowing standard relational operations to query on the dimension tables. It would be interesting to see if the multidimensional indexing techniques proposed for OLAP (e.g., [11] and [8]) can be extended to solve the problem (e.g.,

by implementing the dimension as an index rather than a relational table).

Appendix

Maintenance queries

Alternative A1, daily update:

```
insert into RepairStock as
select  equip, loc, today_date, units_in_repair
from    CurRepInv
```

Alternative A2, daily update:

```
insert into RepairStock as
select equipment, location, today_date,
       sum(qty)
from   Work
where  init_date ≤ today_date and
       complete_date = NULL
group by equipment, location
```

Alternative A2, weekly update:

```
insert into RepairStock as
select  equipment, location, date, sum(qty)
from    Work w, Date d
where (today_date − 7 ≤ d.date < today_date)
  and (w.init_date ≤ d.date )
  and (w.complete_date = NULL or
  w.complete_date > d.date)
group by equipment, location, date
```

Alternative B:
If daily update set
$LAST_UPDATE_DATE = today_date −1;
If weekly update set
$LAST_UPDATE_DATE = today_date −7;

```
insert into RepairOrder as
select  equipment, location, init_date,
        complete_date, qty
from    Work
where init_date > $LAST_UPDATE_DATE or
  (complete_date ≠ NULL and
  complete_date > $LAST_UPDATE_DATE)
```

OLAP queries

q1 for alternative A:

```
select  loc_key, date_key, sum(stk)
from    RepairStock
where (today_date − 7 ≤ date_key < today_date)
group by loc_key, date_key
```

q1 for alternative B:

```
select  loc_key, date_key, sum(qty)
from    RepairOrder r, Date d
where (today_date − 7 ≤ d.date_key < today_date)
  and (r.init_date ≤ d.date_key)
  and (r.complete_date = NULL or
  r.complete_date > d.date_key)
group by loc_key, d.date_key
```

References

[1] R. Armstrong. Data warehousing: the fallacy of data mart centric strategies: short term gain, long term pain. white paper, 1999. NCR Corp.

[2] Bellcore. Stock Inventory Management Module (SIMM): a client-server application from Bellcore for managing plug-in inventory. Product Brochure, 1997. Telcordia Technologies, NJ.

[3] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technologies. *SIGMOD Record*, 26(1), 1997.

[4] C.M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer. In *Proc. of 4th Int'l Conf. on Extending Database Technology*, 1994.

[5] Microsoft Corporation. OLE DB for OLAP programmer's reference, December 1998. www.microsoft.com/data/oledb/olap/spec/.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. of 12th Int. Conf. on Data Engineering*, 1996.

[7] J.M. Hellerstein. Online processing redux. *Data Engineering Bulletin*, 20(3), 1997.

[8] T. Johnson and D. Shasha. Some approaches to index design for cube forests. *Data Engineering Bulletin*, 20(1), 1997.

[9] R. Kimball. *The Data Warehouse Toolkit : Practical Techniques for Building Dimensional Data Warehouses.* John Wiley & Sons, 1996.

[10] Y. Kotidis and N. Roussopoulos. DynaMat: a dynamic view management system for data warehouses. In *Proc. of ACM SIGMOD Conf.*, 1999.

[11] S. Sarawagi. Indexing OLAP data. *Data Engineering Bulletin*, 20(1), 1997.