On the Importance of Tuning in Incremental View Maintenance: An Experience Case Study*

K. O'Gorman, D. Agrawal, and A. El Abbadi

Department of Computer Science
University of California
Santa Barbara, CA 93106
{kogorman,agrawal,amr}@cs.ucsb.edu

Abstract. We originally set out to explore the performance of a particular algorithm for incremental maintenance of a materialized view in a data warehouse. We chose to use a substantial database schema and population, derived from the TPC-D family of benchmarks, with a population size of around one gigabyte. In the process, we discovered that at this size, unexpected issues with the query and update processing had to be understood and dealt with before the results would be meaningful. We discover that incremental maintenance is feasible over a wide range of update sizes (*granularities*), and that in all cases a cursor-based version of the algorithm performs the best. This is a report of both the results of the original inquiry and of some of the events encountered.

Keywords:

View Maintenance, Query and Update Tuning, Distributed Query Optimization.

1 Introduction

Algorithms and frameworks for handling incremental maintenance of materialized views in a data warehouse have been presented largely in the abstract, or tested on small databases[LYC+99]. This work presents the results of exploring (part of) the solution space for implementing incremental maintenance of a view in a data warehouse with a database population of an interesting size (nominally 1 gigabyte).

While the results of this exploration are interesting in themselves, we think that some of the problems we encountered, and the strategies we used, are also interesting. In order to present these, we use a presentation style that is narrative and hence appropriate for describing our experiences in implementing our tests and analyzing our results.

We planned to develop a view query which would assist in answering useful business questions about this data population, in order to explore several issues around materialized views:

* This work is partially supported by the NSF under grant numbers EIA-98-18320 and IIS-99-70700

– Is incremental maintenance substantially better than complete rebuilding of the view, with regard to use of system resources?
– How does the granularity of updates (i.e. the amount of update activity from the data sources that is incorporated into the view in one round of processing) affect the resources consumed by incremental maintenance?

During this investigation, we quickly and repeatedly discovered other issues that demanded our attention. Among these were:

– How the choice of original and update populations affects the results?
– How important is the attention paid to tuning the database; i.e. how does a tuned database compare to a naive approach with regard to view maintenance?
– Assuming that tuning makes a difference, which kinds of tuning make a significant contribution?
– To the extent that tuning involves adjusting coding style, are there clear winners among the various ways to code the algorithms?

The tuning research results we are aware of is focused on individual elements or operations to be optimized, often using techniques or features that are unavailable in commercial DBMS engines. Fortunately, the importance of tuning a database has long been known to the people who operate commercial databases, and there is an extensive literature available from database vendors (e.g. [Ora], [ALS97]) and third parties (e.g. [GC96,Ada99]). One such source with collaboration from a number of vendors is [Sha92].

While they can be helpful, these volumes are on the one hand overwhelming in their massive content, and on the other hand are most useful in an operational context rather than a research context. Blindly following the procedures and advice in these sources could lead to tuning the system specifically to the research tasks, but the goal is to explore performance in a realistic setting. Our approach has been to try for a "middle way" of moderate aggressiveness in tuning.

2 The Data and the Experimental Setting

In this experimental setting one of the key components is the underlying data set that will be used to evaluate view maintenance strategies. We chose the schema of the TPC-D family of benchmarks which has the advantages that it is well-known in the DBMS research arena and that it comes with a generator program dbgen, and a number of interesting queries. The generator can be used to populate the schema and also to create *refresh sets* which are updates, both insertions and deletions, to two of the tables, LINEITEM and ORDERS. Both of these can be done in a wide variety of sizes, including all of the sizes our investigation would require.

Moreover, the schema takes the form of business data, and the queries are accompanied by explanations of the business question that the query attempts to answer. This appears to provide a realistic framework in which to construct a view which would be useful in a real-world setting.

We quickly discovered that in spite of first impressions, the benchmark is not a model of a business. Tradeoffs that were made to achieve a useful and scalable benchmark make the data population an unlikely model for an actual business. As we will explain below, this created some challenges in the conduct of the investigation.

2.1 The TPC-D Benchmark Family

The schema chosen for our tests is that shared by the well-known benchmark suites derived from TPC-D (TPC-H[Traa] and TPC-R[Trab]). It is generated at the desired size by specifying a scale factor SF to dbgen, as a real number indicating the approximate size of the population in gigabytes. The schema is shown in Figure 1 with the number of rows generated for our experiments, i.e. with scale factor 1, resulting in approximately one gigabyte of raw data.

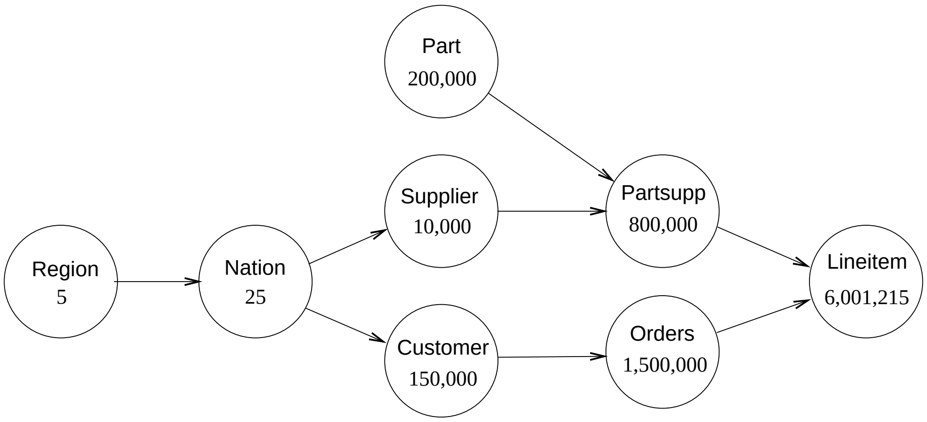

Fig. 1. The schema of the TPC benchmarks. Arrows show the one-to-many relationships. The number of rows in each table is shown for the population size in this study (SF=1).

The benchmarks define 21 queries to be performed on the database, each with a justification (expressed as a business question), and the SQL coding of the query. The SQL contains variables which are to be replaced with randomly chosen constants at each execution of the query. For validation, the benchmark presents the results of each query for one particular choice of values, applied to a database with scale factor of 1.

We selected three of these queries, numbered 7, 8 and 9, because they were similar enough that a single view could be defined which would benefit the execution of all three. Query 7 finds, for two given nations, the discounted dollar volume of shipments made between the two countries in 1995 and 1996. Query 8 finds, for the years 1995 and 1996, the market share of the suppliers of a given nation for all destinations in a given region. Query 9 finds, for a given type of parts, the profit made on that part broken out by supplier, nation and year.

2.2 Data and Workload Characteristics

There were a number of unexpected features of the schema and the updates. For instance, the use of uniform distributions in all attributes means that there is little variation from one supplier to another, or one country to another. For example, shipments from any country in Africa to the USA are exactly as likely as shipments from Canada to the USA.

The data population models a remarkably slow rate of sales for all items. During the seven-year span of the data population, each supplier sells each of its items a mean of 7.5 times, or slightly more often than once per year. We will see the effect this had on the size of the materialized view.

While these attributes make for a good benchmark for database throughput, they were a considerable challenge in terms of constructing a useful materialized view. Aggregating the main fact table on more than a few dimensions tended to produce a view with about the same cardinality of the underlying detail data; this is a direct consequence of the tendency of uniformly and sparsely distributed data not to cluster.

The benchmark defines "refresh" operations which are designed to perturb the database and create a background of activity during the benchmark testing. These operations consist of deleting a number of orders and their associated lineitems, and inserting an equal number of orders and lineitems. The original sequence of order numbers has "holes" which leave room for the insertion of the new orders.

One unexpected feature of the refresh operations generated by the dbgen utility is that the inserted orders and lineitems are *identical to the ones deleted except for the order number*. This is presumably convenient for the operation of the benchmark, since the database activity creates a workload, but at the same time, the activity does not change the results of any query that does not depend on the order numbers involved, as none of the specified queries do.

Of course, this applies to our view query as well, with the result that our initial attempts to use the refresh operations to generate updates to the view resulted in changes which were vacuous. We resolved this by using the first 12,000 insertions and the *second* 12,000 deletions, so that the net effect was not empty. This was not a complete solution, because the resulting insertions were guaranteed to duplicate the quantities in the items we had not deleted, so that we were never inserting new lines in the view. We explored this by performing an additional complete set of test runs with the first 12,000 deletions performed prior to the test proper. In these runs, almost all insertions in the base tables resulted in new row insertions in the view. Tests with the two different starting conditions are labeled as "update/delete" or "update/insert/delete" according to which mix of operations predominates.

2.3 The Warehouse

Our experiments model a warehouse represented in Figure 2. There is a single host housing the data warehouse, and a number of data sources, each holding a single table on which the view is defined. The data sources maintain communication with the warehouse, and when a change occurs at one of the data sources, the change is communicated to the warehouse. At the warehouse, the Update Integrator needs to query (a subset of) the base relations in order to compute the change (*delta*) to the view, in general by issuing queries to the other data sources, and forming a suitable join of the results. When the delta computation is complete, it can be installed in the view. The installation process is the main focus of this paper.

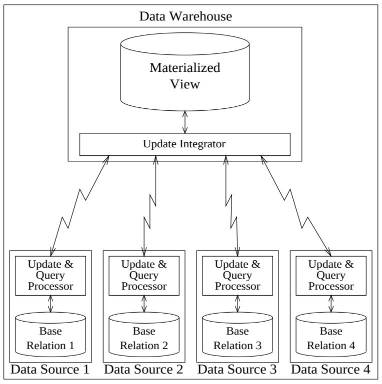

Fig. 2. A model of the data warehouse. Each data source holds one of the tables in the schema. A warehouse over four tables is picutured here.

For our tests, the delta is computed by the method of the Posse[OAE99] framework. Posse is a framework for incremental maintenance in a distributed setting and is representative of many other recently proposed mechanisms[ZGMHW95,ZGMW96,AESY97]. We report the message counts and sizes encountered in that computation, but further consideration of the communication is beyond the scope of this paper. We focus instead on the installation of changes into the view.

3 Materializing a View

We performed all tests on a database of scale 1 (nominally 1 gigabyte), and on five update sets, each update set comprising 96 raw updates, each raw update inserting 25 rows and deleting 25 rows of the ORDERS table and the associated rows of the LINEITEM table. These populations were generated by the dbgen utility that is part of the benchmark suite. The choice of 96 relates to its factorization as 3×2^5 and yields good coverage of the x-axis in a logarithmic plot.

We indexed the tables of the schema with indexes suggested by Graefe[Gra99], with the exception that we did not use a cluster index on the LINEITEM table.

In order to investigate the effects of the granularity of the updates, the updates were grouped into units we called *grains*, collections of consecutive raw updates from among the original set of 96, ranging from a single raw update to all 96. Grains sizes were chosen to be an even divisor of 96, and all the grains of a given size were applied together in a test we called a sample. Each sample run would compute the same function of the same inputs as the other samples constructed from the same update set, a property we used to validate the algorithms used. Five different update sets were used in each sample, and we report average, minimum and maximum times of the five sets for each sample.

Tuning every possible aspect of the system was beyond the scope of our investigation. Here are some of the more obvious possible things that we kept constant.

- We used a page size of 8K.
- We used a constant space layout: system tables were on the boot disk, and all of the tables in our user schema were on the other disk.
- We used unpartitioned tables and indexes.
- We used SQL scripts to drive all operations except for the trigger, which is always compiled in our environment.
- We ran a normal complement of system services, such as inetd and apache, but during all timed tests the network cable was disconnected.
- We allowed the running programs to perform output to the X console and we logged this output to disk, judging the time involved small compared to the intervals being measured.

3.1 The Test Environment

All tests were performed using Oracle $8i$ Enterprise Edition, version 8.1.5.0.1, on RedHat Linux Version 6.1. The machine was a Pionex model M7550MPC-A-V1, which includes a Pentium-III 550MHz CPU. The system includes a modem, sound card, and DVD drive which were unused during these tests. For these tests, RAM was upgraded to 256MB, and a Maxtor 27.2 GB K0192720U IDE hard drive was added to the existing 12.7 GB Quantum Bigfoot drive. An Ethernet card and SCSI controller were also added to the system, but not used directly in these tests.

Tests were performed in the system's normal operating mode, with several servers including an X server running and internet connections active but largely unused. There was, in addition a single (mostly) CPU-bound process running at the lowest possible priority. Tests were run by execution of SQL*Plus scripts, sometimes for days at a time.

All tests reported here were obtained with the same layout of partitions and tablespaces; moreover, identical or functionally similar tables were allocated to the same tablespaces in all tests reported here; different allocations were used initially and were helpful in validating the correctness of the scripts.

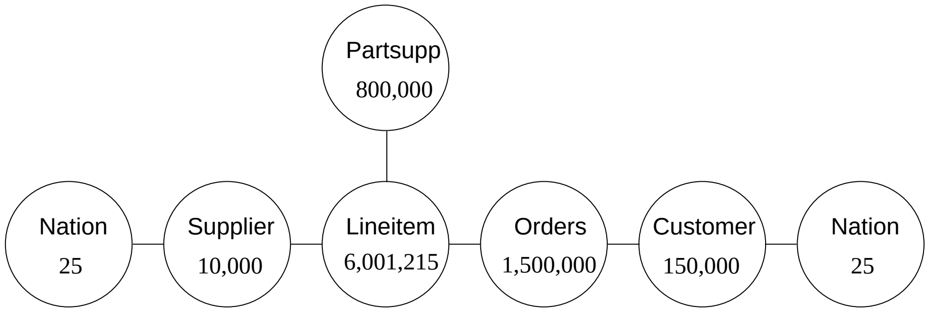

Fig. 3. Join Graph of the View Query. The number of rows for each table is shown. The two instances of the NATION table are treated independently.

3.2 Select and Build a View

We observed that three of the benchmark queries (Queries 7, 8 and 9) had similarities that made it likely that all three would benefit from a single materialized view. For the purposes of the testing, we defined a table named WARE to contain the information from the view. The join graph of the view query is shown in Figure 3. The view abstracts out details of customers, day of month, and delivery date among other things. It represents just the quantity, cost and discounted price of parts moving from one country to another with given months of order and shipping. This view was stored as a table defined by this SQL query:

```
create table WARE as
    select n1.n_name as supp_nation, n2.n_name as cust_nation,
        trunc(l_shipdate, 'MONTH') as shipmonth,
        trunc(o_orderdate, 'MONTH') as ordermonth,
        l_partkey, l_suppkey, sum(l_quantity),
        sum(l_extendedprice * (1 - l_discount)),
        sum(ps_supplycost*l_quantity), count(*)
    from supplier, lineitem, partsupp, orders, customer, nation n1, nation n2
    where s_suppkey = l_suppkey and ps_suppkey = l_suppkey
        and ps_partkey = l_partkey and o_orderkey = l_orderkey
        and c_custkey = o_custkey and s_nationkey = n1.n_nationkey
        and c_nationkey = n2.n_nationkey
    group by n1.n_name, n2.n_name, trunc(l_shipdate, 'MONTH'),
        trunc(o_orderdate, 'MONTH'), l_partkey, l_suppkey;
```

Building this view was the first major database function we attempted in this configuration, so we were not sure how long it will take for view creation. The first few attempts were complete failures, which we tracked down to very flawed query plans being generated by the DBMS, and the resulting exponential growth in the intermediate results. The problem with exponential growth was fixed by having the DBMS obtain statistics on the tables, and giving hints to the optimizer. This resulted in a query plan of nested loops joins. However, we were dismayed when view construction failed for lack of disk space to store the result, after computing for more than 48 hours. We could allocate more space, but even though we expected incremental maintenance to be significantly faster than the build, it was not encouraging to think that operations on data of this size would routinely take multiple days.

Moreover, as we began working with the data, we discovered that a copy of the view could be constructed (using SQL create table as subquery syntax) in about 45 minutes, and the 48 hours became hard to justify. The join graph has just 6 edges, and it was not reasonable that each join should involve work equivalent to copying the result 10 times. This seemed too big by a factor of 3 or so, even allowing for a sort with each join. Since we intended to compare this time with the time for incremental maintenance, we wanted a fair comparison not one that used flawed implementation. Accordingly, we began our first investigation into tuning.

Up to this point, the DBMS had run with all the defaults, as originally installed. After adjustments to buffer sizes and other resource allocations and further manipulation of the query optimizer, it produced a query plan based mostly on hash joins, which executed in about 15.5 hours. This appeared reasonable, measured by our yardstick of 45 minutes to copy the view, and we stopped trying to improve that time. As is typical in database tuning, particularly in the early stages where less than the full job mix is being considered, we quit before we had reached or proved any kind of "optimum", but when the execution seemed "reasonable" and that further tuning was likely to have marginal effect rather than a major one. If in the end, the comparisons we were going to make were close, we would return to the issue and ensure that each of the compared approaches was comparably tuned. This seldom happens in practice and did not in this case either.

The point here is that many factors in a database system effect the running times of algorithms, and some of these factors are difficult to discern at first. We felt it incumbent on us, however, to search these out and adjust them, at least until it became clear that none of the algorithms being evaluated has been unfairly hampered by suboptimal conditions. It also makes sense for our report of the work to make some reference to this effort, so that the reader has assurance that the comparisons are in fact reasonable.

The changes made during this stage were:

- Recode the SQL to project out unused attributes in subqueries.
- Use Oracle facilities to accumulate detailed statistics to inform the query optimizer.
- Recode the SQL to include the CHOOSE optimizer hint, to turn on the cost-based optimizer.
- Alter the SORT_AREA_SIZE parameter from 65536 to 262144.
- Alter the HASH_MULTIBLOCK_IO_COUNT parameter from its default to 1.

– Use the /dev/proc facility to change SHMMAX to 134,217,728.

3.3 Validate the Benefits of Materializing the View

Once the view was materialized, we were surprised that the view table had cardinality almost equal to that of the most populous of the source tables, the LINEITEM table. This was one of the consequences of the design of the population, as mentioned in Section 2. The view query reports sales and shipping data while abstracting away the supplier (representing it with its country) and aggregating order and shipping dates to the year and month. Despite these simplifications, of the 6,001,215 rows in the LINEITEM table, only 2,444 (0.04%) were aggregated in the view, leaving 5,998,144 rows in the view. This was unexpected, and raised the concern that the view might not help the query processing as much as we had hoped.

Query Number	Without View	With View	Speedup
7	603	1	603.00
8	6456	349	18.50
9	3280	240	13.67

Fig. 4. Speedups from the View

In order to validate the usefulness of materializing the view, we compared the performance of queries from the TPC benchmark specification, as adapted to our DBMS (TPC queries) against modified queries altered to use the WARE table (Ware queries). The first test compared the running times of the TPC queries 7, 8, and 9 against the running times of the corresponding WARE queries. As shown in Figure 4, this validated the usefulness of the view under test, since the speedups from using the view ranged from 13 to about 600. These speedups represent the work of precomputing the joins in the view query during view construction, reducing the work required during query evaluation. Of course there is a cost for the speedups, in the form of storage for the materialized view and the activity of maintaining it.

4 Incremental Maintenance of the View

The calculation of the delta for each update used the algorithm from the Posse[OAE99] framework. The calculation proceeds as a series of semijoins, and is followed by installation of the computed change into the view. Installation was explored with SQL and with a cursor-based algorithm.

4.1 Using SQL

The installation of the computed delta was very slow on our first attempts with SQL. The installation was performed in three consecutive SQL commands: an UPDATE of rows already in the view, an INSERT of new rows, and a DELETE of rows no longer needed. The INSERT had acceptable performance from the start, but the UPDATE and DELETE were an order of magnitude slower.

Tuning an Update The original form of the UPDATE was the one corresponding to the English description "update rows in the view which correspond to rows in the delta by adding the values in the delta". This works out to a SQL coding style called "coordinated subqueries". In outline, it looks like this:

```
update WARE
    set WARE.value = WARE.value + (
        select DELTA.value
        from DELTA
        where DELTA.key = WARE.key
    )
where exists (select * from DELTA where DELTA.key=WARE.key);
```

In spite of many trials, we could not make the optimizer generate a query plan that drives the update by the key on the delta. Instead, the update used a plan that tested each row of the view for a matching row in the delta, and altered the ones with a match. All of the nearly 6 million rows of the view are read to perform this operation, and the elapsed time is large in proportion to the size of the view table. Eventually, we even found that this behavior is documented for our DBMS, but we understand that this is the case in at least some other DBMS engines as well.

Eventually, a suggestion from USENET[Lew99] led us to a much better idea: performing an update on a join, in which the optimizer is free to choose the better table to drive the join. This improved the speed of the update by an order of magnitude because the cardinality of the view is at least 1,000 times that of the delta, and for fine-granularity updates, this difference is even more. As modified, the update looks like this:

```
update (
        select DELTA.*, WARE.* from DELTA, WARE
        from DELTA, WARE
        where DELTA.<key> = WARE.<key> and WARE.count=0
    )
set <changes>;
```

The performance of this form is essentially that of a physical index read and a row update for each affected row of the view, because the uniform distribution of the data makes it unlikely that any two rows will share a database block. The point here is that had we not discovered this way to tune the SQL, subsequent comparisons would have been invalid because the SQL would have been unnecessarily impaired.

Tuning a Delete For a view that contains statistical data, it would be feasible to omit the deletion step entirely, and allow tuples to remain that contain only zeroes outside the identifying attributes. We chose to include and measure the deletion step, but tuned the SQL.

Since there is no way in SQL to specify conditional deletion in connection with an update, deletion has to be a separate step. In order to unambiguously identify rows to be deleted, the view contains an attribute count that encodes the number of rows of the LINEITEM table that contribute to the entry in the view. A row can be deleted when this value goes to zero.

Tuning of deletion required considering three ways of formulating the SQL. The obvious way is to test each tuple for a count of zero, but this requires a complete scan of the view. A faster way is to use an index on the count attribute, at the cost of maintaining the index, but with the advantage that only the tuples to be deleted (i.e. those with count equal to zero) will be accessed. A third way involving triggers will be described below. Since only tuples that have just been updated by the delta will have changed, just these tuples can be examined. Formulating efficient SQL for this is a challenge similar to the one for tuning the SQL of the update, and results in a formulation like the following[1], due again to help from USENET[Lew99]:

```
delete from WARE ww
    where (
        ww.<WARE key attributes>
    ) in (
        select
            dd.<DELTA key attributes>
        from DELTA dd
    );
```

4.2 Using a Cursor

No matter how much we tuned the individual SQL commands, they remained three distinct commands, and the optimizer does not apply any global optimization to take advantage of the fact that all three are working with the same data. In order to do that, we were obliged to code a query plan that SQL cannot

[1] Actually, hints to the optimizer are also required, but are omitted because they are not portable. Of course, on different systems it could be that any of these formulations would work differently or require additional or different hints to the optimizer. Portability of SQL really is an illusion.

accomplish, and to do that we used a cursor-based approach. The idea is simple: the process will proceed through the delta one row at a time and make whatever changes are necessary to the view for that row.

The cursor algorithm was expressed in PL/SQL as follows:

```
declare
    deltarec DELTA%ROWTYPE;
    cursor c1 is select * from DELTA;
begin
    for deltarec in c1 loop
        declare
            viewrec WARE%ROWTYPE;
            cursor cview is select * from WARE w
                where w.key = deltarec.key for update;
        begin
            open cview; fetch cview into viewrec;
            if cview%FOUND then
                if viewrec.w0_count = - deltarec.w0_count then
                    delete from WARE w where current of cview;
                else    -- not deleting; must update
                    update WARE wu set wu.values = wu.values + deltarec.values
                        where current of cview;
                end if;
            else
                insert into WARE        -- no viewrec. Must insert one.
                    ( fields ) values ( deltarec.fields );
            end if;
            close cview;
        end;            -- inner block
    end loop;           -- c1 loop
end;
```

The resulting cursor-based code was faster, by about a factor of two, in all tests.

4.3 Update Granularity

We computed and applied our updates in pieces called grains. By varying the sizes of the grains, we were able to explore the effects of granularity on the performance of the several update algorithms. As mentioned in Section 2.2, we used 12,000 insertions and deletions (with respect to the ORDERS table). These were organized as 5 separate updates of 2400 lines each, tested independently in order to explore how execution time might vary from one data set to another, but for some tests fewer than the full 5 updates were tested. Each update could be tested in different granularities by forming grains of sizes that evenly divide

2400; we chose the twelve grain sizes of 25, 50, 75, 100, 150, 200, 300, 400, 600, 800, 1200 and 2400 orders each. The tests were organized in such a way that the exact same result was being computed at each granularity, and only the grain size and the number of grains was changing. This range was an interesting one since it seemed likely at one end that the view tuples being changed would fit in main memory at fine granularity and would not fit at the coarsest granularity. The range from 0.002% to 0.2% of the rows being updated by each grain, seemed reasonable for warehouse updates.

5 Results

We conducted experiments that investigate the cost for incremental maintenance of the view, where updates occur at the ORDERS and LINEITEM tables. We were especially interested in the overhead in computing the results of these changes, and of the instalation of these changes in the view. In figures that present times, the Y-axis is denominated in an arbitrarily chosen time unit that is the same for all tests[2].

5.1 SQL Maintenance Times

Timings were performed on incremental maintenance of the view, for comparison with the time for full recomputation reported in Section 3.2. The worst update time for incremental maintenance was more than 66 times faster than the tuned complete recalculation. Even allowing that the tuning was incomplete (and many DBA's regard all tuning as incomplete), this validated our use of incremental maintenance.

Execution times of the SQL version are shown in Figure 5. For the update/delete instruction mix, for each experimental point shown, five different updates were tested, and the average is shown here along with 95% confidence error bars. The large variability at grain size 150 orders is a feature that persisted when the tests were re-run, and may represent unstable behavior of the DBMS near the inflection point of the graph.

Because these results showed that much more time was expended in installing the delta than computing it, and because the traces for install time were so irregular, we explored the components of the installation process, as presented in panels B and D. Here it is seen that most of the time is spent in the update command, which is the first command of the installation process. In this test, we expect the number of deletes to be half of the number of updates. For the upper panels, the number of actual insertions is minimal. The large amount of time taken by update is caused by the update coming first and doing work that benefits subsequent commands: loading the effected parts of the view table into the buffer pool. Up to about grain size of 150 rows, the buffer pool of the

[2] We are legally prohibited from publishing benchmark results; since these results cannot be used for comparisons to other configurations, they do not constitute a benchmark.

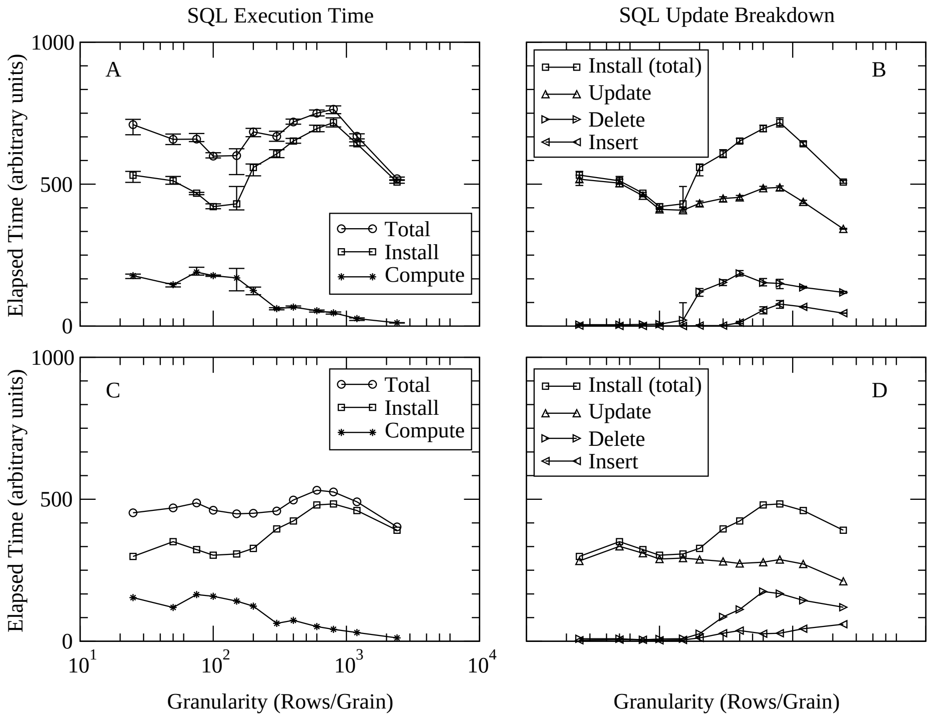

Fig. 5. Execution Time for SQL Version. The upper panels present the results for the update/delete mix. The lower panels present the results for the update/insert/delete mix. The panels on the right break down components of the "Install" trace in the left panels.

DBMS can contain all rows needed for installing the delta, and accordingly the subsequent insertion and deletion stages run very quickly. At larger grain sizes, the buffer pool can no longer contain all rows, and the insertion and deletion times include significant disk I/O. This effect comes later for insertion than deletion, because in this test there were no actual insertions performed, and the SQL could determine this from the primary key without reading table rows. As grain size continues to increase, the times decrease as the reduced number of grains becomes important.

5.2 Cursor Maintenance Times

With reasonable SQL query plans in place, we turned our attention to a different approach to the same problem: using PL/SQL to implement a cursor-based algorithm for installing the deltas. Coding the PL/SQL was entirely straightforward. Figure 6 presents the times for the cursor version of the algorithm. These times are nearly twice as fast as the SQL described above, and somewhat less sensitive to internal features of the DBMS. The computation times were slightly faster because there was no need for an index on the `count` attribute, and thus there was no time spent on maintaining the index. The installation of the delta is faster because this algorithm is a global optimization of the three steps required in the SQL version.

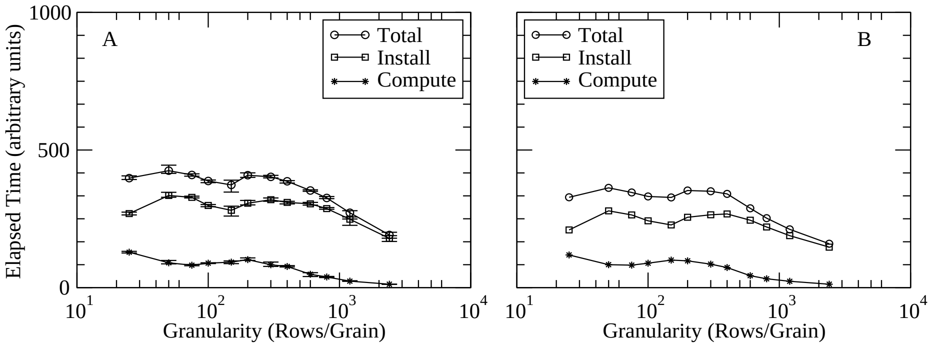

Fig. 6. Execution Times for Cursor Version. Graph A is for the update/delete mix. Graph B is for the update/insert/delete mix.

In this test, the times for the SQL and cursor versions of the algorithm are closer than they are where additions duplicate information already in the view. This is mostly due to a faster execution in the `update` command, which has less work to do. It is not clear why the work is not transferred to the `insert` command, but that does not appear to happen.

5.3 Return to SQL: Using Triggers Instead of an Index in SQL

Since the cursor-based algorithm was so much faster than the SQL, we became concerned that we may not have used the best possible formulation of the SQL commands. We therefore returned to the SQL and reconsidered the design. Cycling between alternatives is common in tuning, when results in one approach can shed light on another.

Being aware that the SQL times were hampered considerably by the need to maintain an index on the count attribute to support deletion, we searched for a solution without this index. The cursor version was able to dispense with an index because it has the concept of a *current row* and could deal with deletion as each row was updated. Without other modifications, dispensing with the index in SQL would force the DBMS into a complete table scan of the WARE table, which would have been disastrous for performance. We needed a way to inform the SQL DELETE of the subset of rows to be considered, without the overhead of maintaining an index. We tried to modify the DELETE command so that it would use a query plan driven by the tuples in the delta, and test corresponding tuples in the view for a count of zero. We were unable to coax the optimizer into such a query plan. Instead, we implemented a solution using an auxiliary table maintained by a trigger that activated on updates to rows of the view.

The trigger was set to "fire" when any row was altered by an UPDATE command resulting in a count of zero. The action performed was to add a row to an auxiliary table, which table was subsequently used to drive the DELETE. This worked well, and we found the total times for the SQL using this approach were much closer to the times for the cursor version, especially at small-to-moderate grain sizes. Figure 7 presents the results.

5.4 Comparison of Results

We have explored the times taken by three versions of the maintenance algorithm under two different mixes of operations. These times are presented together in Figure 8. The cursor-based version is a clear winner, although at some granularities the trigger-based SQL version is reasonably close.

One unexpected feature of the execution times is that the times vary so little. With the grain sizes varying by two orders of magnitude, a much larger variation might be expected. We conclude that in many cases the choice of quite small updates may be a reasonable way to keep the update window short, and reduce the length of time during which records may be locked by update processing.

6 Conclusion

We have presented our experience with performing timing testing of view maintenance for a warehouse on a commercial DBMS, and some of the pitfalls for the unwary. Returning to the issues set forth in Section 1 in terms of tuning, we discovered

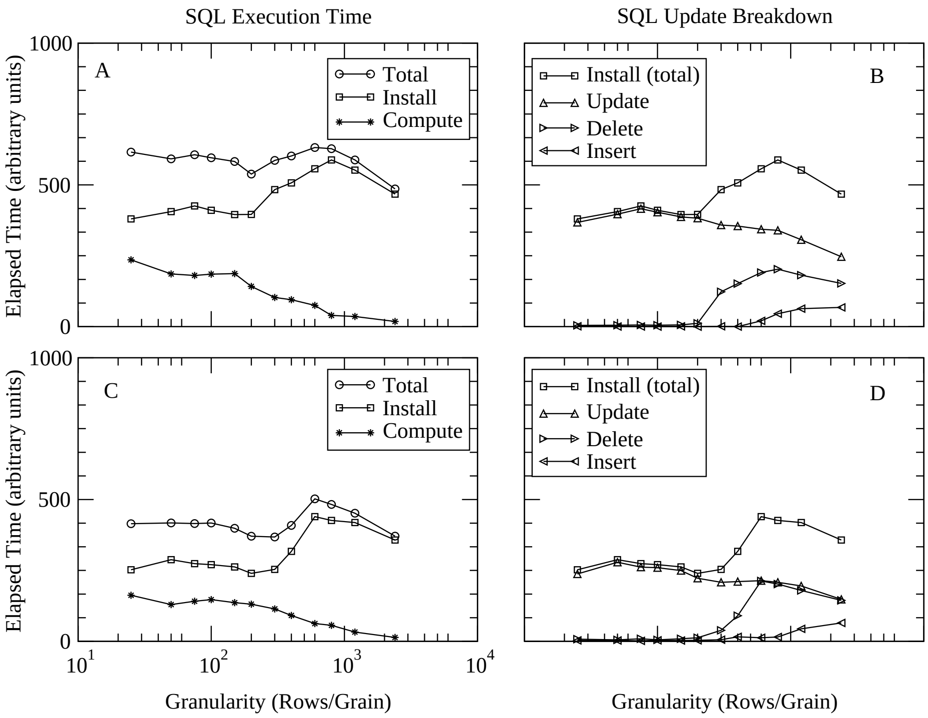

Fig. 7. Execution Time for SQL Version With Triggers. Panels A and B present the update/delete mix; Panels C and D present the update/insert/delete mix.

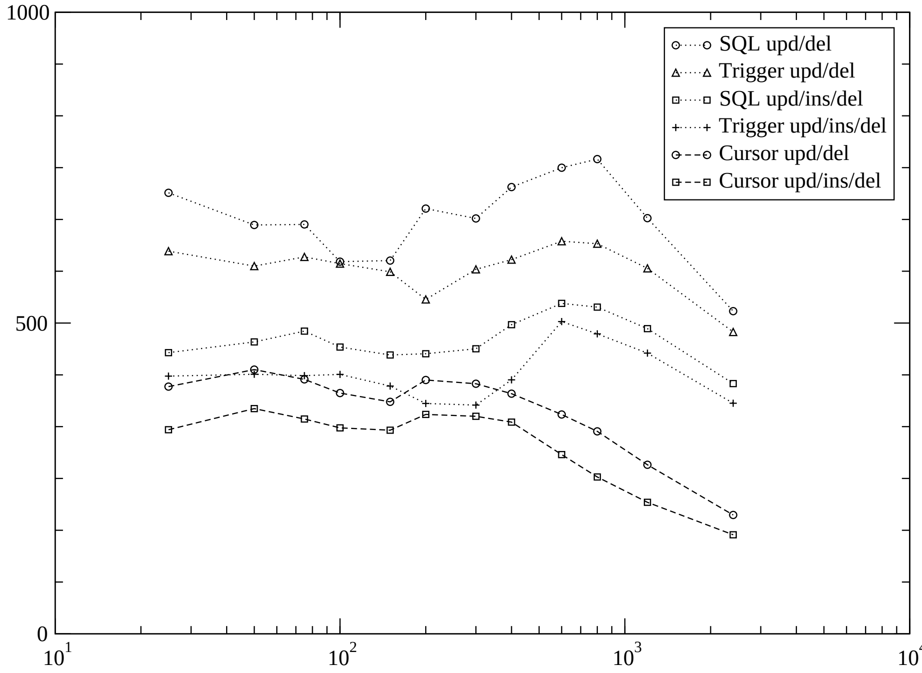

Fig. 8. Execution Times for Each Algorithm

- Some of the characteristics of our chosen data set that make it good for a benchmark created difficulties for our testing. Our overall impression is that a dataset from a real business would have seen greater benefit from tuning and from the view than is seen here.
- Comparing our tuned algorithms, at some granularities we find that cursor-based updating may be as much as four times faster than SQL with an index-driven DELETE, and somewhat faster than a trigger-based SQL solution. The untuned versions were sometimes so bad we were unwilling or unable to complete timing tests. We conclude that comparing untuned algorithms is meaningless, and running them is not sensible.
- We saw improvements from a number of tuning techniques. There was no clear winner. One surprise was that the most efficient SQL for a given task may be quite different from the first formulation that comes to mind, although they may be theoretically equivalent.
- A version using cursors was the clear winner for incremental maintenance.

Regarding the lessons for incremental maintenance, we concluded

- Incremental view maintenance appears feasible at all the grain sizes we tested, although naturally larger grains lead to longer periods with locked data. The overall time was surprisingly uniform; it appears that economies of scale were offset by cache effects to some extent. We conclude that where other considerations allow, the short update window that goes with small update granularities may be attractive.

– Incremental maintenance is at least an order of magnitude faster than complete recomputation of the view. At one gigabyte of data, complete recomputation is not feasible on more than a very occasional basis.

References

[Ada99] Steve Adams. *Oracle 8i Internal Services for Waits, Latches, Locks, and Memory*. O'Reilly, 1999.

[AESY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance in Data Warehouses. In *Proceedings of the 1997 ACM International Conference on Management of Data*, pages 417–427, May 1997.

[AISS95] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation . In *7th Annual Symposium on Parallel Algorithms and Architecture (SPAA '95)*, 1995.

[ALS97] Eyal Aronoff, Kevin Loney, and Noorali Sonawalla. *Advanced Oracle Tuning and Administration*. Oracle Press, 1997.

[GC96] Mark Gurry and Peter Corrigan. *Oracle Performance Tuning*. O'Reilly, second edition, 1996.

[Gra99] Goetz Graefe. The Value of Merge-Join and Hash-Join in SQL Server . In *25th VLDB Conference, Ediburgh, Scotland*, 1999.

[Lew99] Jonathan Lewis. Usenet postings and private emails. URL http://www.jlcomp.demon.co.uk, 1999.

[LYC+99] W.J. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. Technical report, Stanford, October 1999.

[OAE99] Kevin O'Gorman, Divyakant Agrawal, and Amr El Abbadi. Posse: A framework for optimizing incremental view maintenance at data warehouses. In Mukesh Mohania and A. Min Tjoa, editors, *Data Warehousing and Knowledge Discovery First International Conference, DaWaK '99*, number 1676 in Lecture Notes in Computer Science. Springer, August/September 1999.

[Ora] Oracle Corporation. *Oracle 8i Tuning*. Part No. A67775-01.

[Sha92] Dennis E. Shasha. *Database Tuning: A Principled Approach*. Prentice Hall, 1992.

[Traa] Transaction Processing Performance Council. *TPC-H Benchmark Specification, version 1.1.0*. URL http://www.tpc.org/rspec.html.

[Trab] Transaction Processing Performance Council. *TPC-R Benchmark Specification, version 1.0.1*. URL http://www.tpc.org/hspec.html.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *PDIS 96*, December 1996.