Don't wear your OLTP hat when dealing with mass inserts, updates, and deletes or you are asking for trouble. Jack Parker offers several alternative approaches that will cost you a lot less time and trouble.

© 2002 International Business Machines Corporation. All rights reserved.

Introduction

With the relatively recent advent of decision support systems (DSS) in the UNIX®/RDBMS world, database professionals are faced with challenges that mainframe programmers have been addressing for decades. We have been writing OLTP applications for so long that we bring that mindset and those methods with us when we enter the DSS world. This does our hardware and databases an injustice. They are capable of much better things.

In the OLTP world, we might be concerned with a single user entering a customer order. We want to make sure that the line-item table, the order table, and, possibly, the inventory and customer tables are all updated in unison. It does us no good if the order is updated to reflect the price of the item if the line-item update fails. We know the customer ordered a $5.00 item, but we don't know what it was. To guarantee that these updates are handled properly, we use transactions to ensure that all operations are performed (or not) in unison. We worry about transaction logs so we can redo or undo work. Our database software has an entire layer of overhead devoted to supporting this method of processing. Our thought patterns are locked into a "transactional methodology."

In the DSS world we need to perform operations against massive amounts of data. We don't have individual users issuing updates against singleton rows or sets of rows. Instead we perform millions of inserts or updates, maybe even a few million deletes.

If we come to the DSS world with a transactional methodology, we are tying lead chains to our feet.

Initial testing of the techniques in this article were accomplished on a Sun 6500 with 16 processors, 16GB of Ram and 1.2TB disk. Final testing was accomplished on a Sony 133Mhz PentiumII with 500MB of RAM and 40GB of disk. Interestingly enough, both machines behaved
comparably after scale had been factored out. The reader will find references to extrapolation in timings within this article. These values were reached mathematically and reflect the performance that may be found on either a machine.

**Example 1: DSS inserts**

In a recent case, I was handed a process that loaded a 1 billion row table with new data: inserts. Duplicate data was undesirable. To prevent this, the developer attached a unique index to the table and used the IBM Informix® Extended Parallel Server™ (XPS) parallel loader in deluxe mode (in which the loaded rows are subject to all validation criteria) to reject duplicates. The load process ran at about 1 million rows per hour. Monthly, when the process had more than 10 million rows to insert, the logical logs would overflow and the engine would start to roll back all of the inserts. Because rollbacks take substantially longer to undo operations than the original operation, this meant a further 20 to 30 hours in rollbacks. In other words, there would be 30 to 40 hours devoted to a process that failed.

**Why this approach doesn't work**

To understand why this approach is disastrous let's take a closer look at what the engine does:

1. The Loader reads a record from the input file (or files).
2. The record is converted into the internal format (it is now called a row).
3. An index lookup is performed to see if the index for this row already exists. Because this is a large table and the index is six levels deep, this means, potentially, six reads per index lookup.
4. If the index is not found, the row is appended to a page with space on it; that page is therefore read (or may still reside in the buffer from the last insert).
5. This may mean allocation of more extents for this table.
6. A "before" image of this page is preserved in the physical log (before any changes)
7. The insert is recorded in a transaction log.
8. Periodically these data pages are flushed to disk, either through a checkpoint or a foreground write.

In a development environment with much less data, this process would have never overrun the logs, index lookups would have been much quicker, and the number of rows to insert would have been smaller. The entire procedure might have run in 20 to 30 minutes, which is quite acceptable. It is not until we run into a large table that we start having problems.

In the transactional methodology, we want to have the most recent data read or written held in a buffer, not only so that writes can be grouped together and written more efficiently, but so that the data that is requested will generally be in that buffer. To this end we allocate a substantial portion of our memory to these buffers and we worry about least recently used (LRU) queues to manage them. We seek to have a buffer read hit rate of well over 90 percent and a cache write rate of higher than 80 percent. This works well when we are working with a small set of rows; however, this presents a layer of overhead which slows things down when dealing with volume. This layer of overhead is sometimes known as the "buffer wall."
Avoiding the "buffer wall": Light scan and light append

The gating factors for our insert are:

- We are trying to insert data through this buffer wall.
- We are doing index lookups for each row we intend to insert.

If we could somehow speed those up or avoid them, we could move the data along faster.

Fortunately there are the Light Scan and the Light Append features. In a Light Scan, the data read from the table is stored in its own private buffer pool, with the understanding that there will be no need to retain the data for further operations. There is an inherent implication that the entire table, or at least an entire fragment of the table, will be read. With a Light Scan, the engine doesn't flood the buffer pool with the contents of a large table and avoids the overhead of buffer management. A Light Scan can be significantly faster than a traditional scan.

The Light Append functions in a similar way but in reverse. There is no attempt to buffer the data in the buffer pool, it is written directly to new pages. At the end of a successful load the new pages are appended to the existing table. With a Light Append you can expect a minimum load rate of 2 GB per CPU, per hour.

To obtain optimal performance with these features you must be careful to fragment your tables across multiple dbspaces. In this first example, the 1-billion-row table is fragmented across 32 dbspaces. The temporary tables can also be pre-created and, according to the size required, fragmented across multiple dbspaces. It is best to keep any given table under 500 MB per dbspace, however this is not always feasible. In this particular case, our single table consumed 2 GB per dbspace. Hence, our scan time of 15 minutes instead of 4 minutes. Different instances have different business requirements and hence different guidelines. In this case we set up dbslices with dbspaces matched to the number of processors. (Under IBM Informix Extended Parallel Server dbspaces are collected into entities known as a 'dbslice'. This provides for a larger unit of administration (a dbslice). It also allows the DBA to specify fragmentation at a more removed level "fragment by hash(column) in dbslice1. So our 16 processor machine had a single 32 dbspace dbslice, a 16 dbspace dbslice, an 8 dbspace dbslice, and a 4 dbspace dbslice. Temp space was spread across a 16-dbspace dbslice. A maximum ratio of 3 dbspaces per processor is the goal. If we had had the disk we could have increased the larger dbslice to 48 dbspaces which would have cut scan time by one third.

Avoiding index lookups by using hash joins

This ability to read and write data quickly means that we can redesign our applications to take advantage of this speed and obtain marked performance improvements. If you read the companion article "Tuning DSS Queries," you will note that it makes extensive use of the hash join. A hash join can join data from two tables at a rate roughly one to three orders of magnitude faster than an indexed join. It is a powerful tool, but should be used only when it is desirable to join the contents of an entire table or a large portion thereof.

So let us approach our problem from above again, avoiding indexes and the buffer wall.
Instead of probing the destination tables index pages for each row to be inserted, we will use a hash join to determine if there are any matches (duplicates) between the new data and the destination data. We will then act on that knowledge.

1. The first thing to do is get rid of the index on the destination table. If the index exists, the optimizer may choose to use it instead of our preferred method which is the hash join.

2. Let us then load our new data into a table where the optimizer can see it. We create a raw table which duplicates the final destination table, and an external table 'sameas' the raw table. The load of 10 million rows into that table is a matter of under 60 seconds. We could logically have not loaded the new data. XPS will treat an external table as though it were a real table; however, the optimizer would have assumed there were 2 billion (MAXINT) rows in this file and scanned it last, building the hash table from the larger table, which is too big to fit into memory. This would have caused a memory overflow condition and required an hour or more of swapping data to and from temp disk.

3. We update statistics for the new raw table so that the optimizer can understand how many rows are in it. LOW is plenty for what we need. In the interests of brevity we will not repeat an UPDATE STATISTICS after every table load or table insert herein, but this is a recommended addition.

4. Next we match the new table against the old table:

   ```sql
   select raw_table.*
   from destination_table, raw_table
   where destination_table.key = raw_table.key
   into temp t1 with no log;
   ```

   We could cut another few seconds off of our time by pre-creating this "t1" table and fragmenting it across multiple dbspaces. I refrain from doing this throughout the article only to keep it simple.

   Scanning the entire destination table may seem an incredibly expensive thing to do, but we have fragmented the table over 32 dbspaces and, thanks to the Light Scan, this hash join takes 15 to 17 minutes.

   Any duplicates will be inserted into the temp table (t1). Our first problem now asserts itself: we need to know if any rows were inserted to that table (t1) and take different actions depending on that knowledge, a conditional. SQL doesn't support what we need very well. We can, however, easily pull this sql into a script, perform the checks there and branch accordingly.

   If no rows were inserted into the temp table, our source data is pure and we can just:

   ```sql
   insert into destination_table select * from raw_table;
   ```

   On the other hand if rows were inserted into t1 we have to find out which rows are duplicate and not insert them. We therefore:

   ```sql
   insert into t1 select * from raw_table;
   ```
We have now appended the smaller of the two datasets (the inserts) into our t1 table. This created duplicates where they are easy to find:

```sql
select key
from t1
group by key
having count(*) = 1
into temp t2 with no log;
```

In t2 we now have the good unique data. We can join back to the raw_table from there to load directly into the destination_table:

```sql
insert into destination_table
select raw_table.*
from t2, raw_table
where t2.key=raw_table.key;
```

We could have combined these two steps, grouped by the entire row, and inserted into the destination_table. This would have avoided the hash join. In tests, that yields a 30-second improvement; this will vary depending on your tables.

If we want to go one step further and update the destination table with the duplicates we would also:

```sql
select key
from t1
group by key
having count(*)=2
into temp t3 with no log;

select raw_table.*
from t3, raw_table
where t3.key=raw_table.key
into temp t4 with no log;
```

We could also have combined these two steps, grouped by the entire row, and inserted into the t4 table. This would have avoided the hash join. In tests, that yields a 1-second improvement; this will vary depending on your tables.

From there we can update the destination table directly from the temp table t4.

Table 1 shows how much time we actually spent inserting rows using this method.
Table 1. Time to insert ten million rows into a one billion row table

<table>
<thead>
<tr>
<th>Action</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our initial load of data (let’s be generous)</td>
<td>60 seconds</td>
</tr>
<tr>
<td>Hash join between raw and destination</td>
<td>16 minutes (15 to 17 was the average number)</td>
</tr>
<tr>
<td>No duplicates insert</td>
<td>20 seconds</td>
</tr>
<tr>
<td>Duplicates resolution</td>
<td>90 seconds</td>
</tr>
<tr>
<td>Update with duplicates (an update join is a hash join with a subsequent update-more later)</td>
<td>17 minutes</td>
</tr>
</tbody>
</table>

In short, our process now runs successfully in under 20 minutes, as opposed to sometimes running in 10 hours and sometimes failing in 30 to 40 hours. If we want to do the updates as well, we can still count on a total time of under 40 minutes. The best part is that even if we have 30 million rows to load, the time doesn’t change much: the gating factor is the scan of the destination table of 15 (or so) minutes, not the incoming data.

This sort of algorithm can be applied to just about any situation, as long as you understand the rules show Table 2:

Table 2. Rules for thinking about DSS processing

<table>
<thead>
<tr>
<th>Do not</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not think in terms of rows. As long as you think of read a record, read a row, check something, and insert/update the row, you are doomed to the transactional methodology.</td>
<td>Think of rows as a set of data. Work with all of the rows at once.</td>
</tr>
<tr>
<td>Do not use transaction logging. Transaction logging introduces an expensive layer of overhead. Not only does it slow things down, it imposes limits on what can be done before overflowing a log somewhere. Additionally, if a process fails halfway through, the last thing we want is to figure out where in the process it failed and then restart it. It will generally be less expensive to restart the process. Consider that a restore and reapplication of updates in the case above would have taken 10 to 11 hours, and is a rare occurrence. In the following examples, we will use some rewrite techniques that provide us multiple opportunities to step back to a previous state and try again, should things go bad.</td>
<td>Throw out all of your indices (we’ll talk more about this later).</td>
</tr>
<tr>
<td>Do not use database-level referential integrity. In the case above, we used a hash join instead of a unique index to prevent duplicate data from entering our table. We can approach most referential integrity challenges in the same manner to the same effect.</td>
<td>Combine your data into single tables. Joins are expensive; if all of the data you need is in a single table, it becomes a lot easier to work with.</td>
</tr>
<tr>
<td>Do not open a cursor against a table. In so doing you are taking all of the power of the engine and passing it through a little pipe to your process. Do all of the work in the engine itself. There may come a time when a cursor is the only tool available to handle your situation, understand the implications before using it.</td>
<td>Extract sets of rows that you want to work with, work with them, and then put them back or into a new copy of the table, to which you append the other unchanged set of rows.</td>
</tr>
</tbody>
</table>

Example 2: DSS updates

There are many different ways to skin this beast using a DSS methodology. There are fewer options in an OLTP world. In the OLTP world, we will read an input from somewhere, find the matching row in the destination table, make changes to that row, and write it back to the database.
This method tends to be slow and you can count on a performance rate in the tens-of-thousands per hour. Against the 1-billion-row table mentioned above, 20 million updates are applied to the table every night. An OLTP methodology for this process would never catch up to the amount of data to be updated.

**Alternative 1: Update join**

Fortunately IBM Informix Extended Parallel Server provides an "Update Join."

```
update Destination
set destination.col1 = input.col1
from destination, input
where destination.key = input.key
```

**Alternative 2: Additive update**

Let us assume, though, that we don't have the Update Join and approach this in a different way. Instead, let us apply 20 million updates to a 100-million-row table. Our first step is to turn the 'update rows' or input into rows that look exactly like the destination rows. This is done while loading the data into a temporary table. We can then do a number of things, all of which involve writing a new copy of the table:

I will abbreviate syntax to just show the interesting parts, especially since I don't know what we're updating here. I have avoided aliasing to make things more readable.

If the updates are additive (update destination set col1=col1+foo.col1...) we can:

```
create raw table new_dest
create raw table load_table sameas destination
insert into load_table
select * from input_files;
insert into new_dest
select keys, destination.col1 + load_table.col1
from old_destination, outer load_table
where old_destination.key = load_table.key
```

Note that you cannot use sameas syntax when creating a raw table. This syntax is reserved for external tables.

This may seem counterintuitive -- how can an outer join be more efficient than an update? It is probably slower than an update join (see Table c), but we've replaced an index strategy with a strategy that uses reads and writes. As mentioned previously, an IBM Informix engine has the power to rewrite entire tables faster than conventional databases can handle an indexed update of this size, including IBM Informix databases.

**Alternative 2: Replacement updates**

If the updates are replacements, our strategy varies slightly. We use an indicator attached to each row to indicate if it should be replaced with the new data or not.
1. First let's build a temporary table and mark all of the old rows with an indicator of 0:

```sql
create table temp_table keys, indicator smallint;
insert into temp_table
select keys, 0 indicator
from destination;
```

2. Next let us insert all of the new keys and an indicator of 1, into the same table:

```sql
insert into temp_table
select keys, 1 indicator
from input_files;
```

3. Now, we pull all of the keys for our final destination into a new temporary table t2. Those that have an indicator of 0 came from the destination table, and those with an indicator of 1 came from the input_files table. Those that are unchanged (indicator=0) can be put directly into a new copy of our destination table.

```sql
select key, max(indicator)
from temp_table
group by key
into temp t2 with no log;

insert into new_destination
select destination.*
from destination, t2
where key=key
and t2.indicator = 0;
```

4. Our replacement data can now be appended directly from the input_files (external) table:

```sql
insert into new_destination
select * from input_files;
```

5. Finally, we drop the old table and rename the new table:

```sql
drop table destination
rename table new_destination to destination;
```

This method may seem tortuous. We've had to scan the destination table twice: once to get all of the keys, the second time to get all of the data. We've also built a potentially very large temporary table which may also be expensive to scan. The point here is that the speed differential between full table scans and re-writes can still be much faster than an indexed update.

Table c shows some times for these methods updating 100 million rows with a 20-million-row input table:

**Table 3. Elapsed time for updating 100 million rows**

<table>
<thead>
<tr>
<th>Method</th>
<th>Elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Time</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>OLTP update</td>
<td>more than 100 hours (extrapolated)</td>
</tr>
<tr>
<td>Update join</td>
<td>30 minutes 48 seconds</td>
</tr>
<tr>
<td>Additive update</td>
<td>17 minutes 54 seconds</td>
</tr>
<tr>
<td>New table update</td>
<td>67 minutes 19 seconds</td>
</tr>
</tbody>
</table>

Notice that the additive update is actually faster than the update join. This is because it doesn't have to rewrite pages. It's doing all of its work with Light Scans and Light Appends. The actual update time is only slightly longer than the time to scan the two tables.

There are variations on this method. The key to it is that no update actually takes place; we avoid the buffer wall to perform all of these operations. At the end, if the process succeeded in all steps, we can drop the original table and rename the new_destination table. There is, however, an implication that we are going to update a large portion of the table. If the number of updates to apply is small, then by all means stick with an OLTP method, provided you don't have XPS and the update join.

You will note above, that the XPS update join is not as fast as might be expected. The actual scan and join of these two small tables should take under twenty minutes. The rewrite of the updated pages is what is costing us time.

You will also note that by rewriting the table with new updates as above, we have not lost data. If we do a count of what we expect before the update and what we expect after the update, we can check both tables in question and decide whether they are okay before dropping our old table. Hence we don't require logging to recover from a problem.

The new table update should have run faster (45-minute time frame), except that it overflowed memory on the larger hash join. This is a serious flaw with the DSS methodology which needs to be understood.

**Memory**

Memory allocated to a hash join comes out of DS_TOT_MEMORY. The amount of memory allocated is tied directly to the PDQPRIORITY setting. If you have a PDQPRIORITY setting of 80 (and MAXPDQPRIORITY=100) then you will get 80 percent of the memory available. MAXPDQPRIORITY is also a percentage, so your effective PDQPRIORITY setting is MAXPDQPRIORITY/100 * PDQPRIORITY/100 * DS_TOT_MEMORY.

In XPS 8.31 this has changed slightly in that PDQPRIORITY and, hence, memory is allocated dynamically as required; however, this dynamic allocation is still not perfect. The engine builds a hash table in memory from the smaller of the two tables in the join, and then probes that table with the larger of the two tables. If there is insufficient memory to build the hash table, portions of that table are swapped out to temp disk. Even with temp disk this is generally fine, although if your query overflows by more than 200 percent (100 percent in memory and 100 percent on disk) then the swapping requirements will slow things further.
You can figure out how much memory you will need for the hash table with the following formula. A hash table entry is 32 + keysize + rowsize (of the smaller table). Hence if you can make your smaller table thin (low value for rowsize) and (or) join on a small key, the join will require less memory and will run faster. Understanding this and keeping an eye on it is a key to tuning a DSS process.

Indices (or indexes, if you prefer)

When you create an index with the expectation of using it, you are trying to do one of two things:

- Provide quick access to individual rows.
- Provide all the necessary data for an operation in the index itself.

When using a DSS methodology, the index can fool the optimizer into using a nested loop join instead of a hash join. This is especially treacherous for large tables, since finding the row may mean traversing several index pages, depending on the depth of the index. If you are working with a lot of data, you will be filling the buffers with pages from which you will pull one or two rows; you may have to read some pages multiple times, potentially doubling the work to be performed. If you are using an index, expect a throughput that can be measured in tens-of-thousands per hour. There is nothing in a DSS methodology that requires an index.

Of course, there are situations in which an index can be invaluable. If we need to delete 200 rows from a large table, it may be much quicker to build an index on the table and then issue 200 individual update or delete statements than to use a DSS method.

Example 3: DSS delete

Alternative 1: Delete join

IBM Informix Extended Parallel Server offers a companion to the update join for deletes, known as the delete join. In version 8.30 you could not alias the tables; this was cleaned up in version 8.31. The syntax is:

```sql
delete from destination
  using destination, input
  where destination.key = input.key;
```

This is a very effective way to apply many deletes against a table. Again, this is a hash join in which updated (deletes applied) pages are rewritten to disk on the back end of the hash join. However, you should consider what you are doing before running such a command. A delete creates a hole in a page; this hole would be available to be refilled by an insert statement or, in some cases, an update statement in an OLTP environment. However, we have worked hard to avoid such statements; the application of many deletes to our DSS table will leave it full of holes that will not be reused. If there are enough such deletes we will want to rebuild the table.

Alternative 2: Rewrite approach

Let's take a look at a delete using a rewrite approach instead of a row removal approach.
Essentially, what we want to do is write a new table excluding the rows we don't want to keep. This might be thought of logically as:

```sql
insert into new_table
select * from old_table
where key not in (select key from delete_table);
```

This would be a disastrous way to approach the problem because a NOT IN condition will result in a nested loop join between the two tables. A better approach would be:

```sql
insert into new_table
select * from old_table
where not exists (select 0 from delete_table
  Where delete_table.key=old_table.key)
```

This will force a hash join between the two tables.

### Table 4. Extrapolated elapsed times for deleting from a 1-billion-row table

<table>
<thead>
<tr>
<th>Delete</th>
<th>100 KB rows</th>
<th>10 million rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete OLTP</td>
<td>approximately 60 minutes</td>
<td>more than 10 hours</td>
</tr>
<tr>
<td>delete join</td>
<td>approximately 17 minutes</td>
<td>approximately 60 minutes</td>
</tr>
<tr>
<td>'not exists'(rewrite)</td>
<td>approximately 30 minutes</td>
<td>approximately 29 minutes 45 seconds</td>
</tr>
</tbody>
</table>

Important: The times shown in Table 4 are extrapolated. They express the sort of times you should expect from these different methods, not actual times. Times for "delete not in" are not shown because when I tested this under XPS 8.31 the optimizer changed it into a hash join. Online, IDS, or IIF might be expected to treat this as a nested loop.

- The OLTP Delete performance is a factor of how many rows there are to delete times the duration of each delete, as the number of rows to delete increases, the time to perform these deletes increases linearly.
- The Delete Join performance is a factor of the time for a hash join coupled with page rewrites, the time to perform these deletes increases to some degree as the number of rows to delete increases as more pages may potentially need to be re-written.
- The Rewrite Delete performance is also a factor of a hash join, but coupled with light append writes instead of page rewrites. Since the entire table is being rewritten each time, the time to execute this statement will DECREASE as there are more rows to delete because there are fewer rows to write.

Not only is a table rewrite potentially faster than an OLTP join or a delete join, it leaves our table in a compact and efficient state.

### Conclusion

Obviously, this sort of methodology requires some extra disk to play around with for second copies of tables and so forth: plan on it. Disk space is a lot cheaper than processing time-- there are only 24 hours in a day to process; if spending $100,000 on disk space will allow you to push more data
through, you've saved the cost of the much larger computer that would be required to handle the
data using a transactional methodology. If you are short on database space you can use external
tables as substitutes. Just be aware that the optimizer does not know how many rows are in an
external table.

We have covered three basic DSS situations here: insert, update and delete. You will no doubt run
into others situations not covered here. If you follow the guidelines laid out above you will come up
with innovative old ways of doing things. When you do, write me, I'd like to hear about them.

When I say "old," those of you who have been around a while may recognize some ancient
mainframe concepts hidden in here. The IBM Informix Extended Parallel Server engine is well
suited to perform these feats of DSS processing. These methods will work with other RDBMs, but
with IBM Informix Extended Parallel Server they will scream.