A Cross Comparison of Data Load Strategies

Anita Richards
Teradata
Talking Points

- Data Maintenance Requirements
- Load Time Breakdown
- Transformation and Cleansing
- “Updates”: Load Utilities and SQL
  - How they work
  - Performance Characteristics
    - including base table load and index, fallback maintenance
  - Maximizing performance
- Load Strategies
  - Performance Comparison of Load Techniques
  - Strategies for Real-Time Availability
  - Strategies for Minimal Load Times
Disclaimer

• Rates shown achieved on
  • 2x5200 (550MHz)
    • Extrapolate rates up 9% for current 5300 nodes.
  • 1 WES (Ver. 3.0) Array Cabinet,
  • 80 10K RPM drives, RAID1 (mirroring)
  • 20 AMPs,
  • Client: 4400 (450MHz),
  • Client to Teradata DBMS: Dedicated 100 MB private LAN.
  • Teradata V2R4.x
  • Controlled Environment.
  • Our SQL and Database Demographics.

Your rates will vary from our rates.
Data Load Requirements

• What’s the Goal?
  • Real-Time Data Availability
  • Minimized Delay Data Availability
  • Minimized Load Times
    • Archival data loading

• How does workload mix impact data load requirements?
  • Dirty Reads?
  • Lock Contention?

Real-Time Availability
Low Update Rates
(Delayed Availability)
Minimal Load Time
High Update Rates
Data Elimination Requirements

• What’s the Goal?
  • Minimize Data Storage
    • Solution: MultiLoad Delete Task
  • Minimize Query Response Times
    • Archive table separate from Active table
Load Time Breakdown

• End-to-End Time to load includes
  • Receipt of Source Data
  • Transformation & Cleansing
  • Acquisition
  • Target Table Apply
  • Fallback Processing
  • Permanent Journal Processing
  • Secondary Index Maintenance
  • Statistics Maintenance
**“Update” Utilities and SQL**

<table>
<thead>
<tr>
<th>Update Method</th>
<th>Receipt of Source</th>
<th>Transform</th>
<th>Acquisition</th>
<th>Apply</th>
<th>Fallback</th>
<th>Permanent Journal</th>
<th>Secondary Index, etc.</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWB Load (FastLoad)</td>
<td>☺☺</td>
<td>☺</td>
<td>☺</td>
<td></td>
<td>☺</td>
<td>☺☺☺</td>
<td>☺☺☺☺☺</td>
<td></td>
</tr>
<tr>
<td>TWB Update (MultiLoad)</td>
<td>☺☺</td>
<td>☺</td>
<td>☺</td>
<td>☺☺</td>
<td>☺</td>
<td>☺☺☺</td>
<td>☺☺☺☺☺</td>
<td>☹</td>
</tr>
<tr>
<td>TWB Stream (Tpump)</td>
<td>☺☺</td>
<td>☺</td>
<td>☺</td>
<td>☺☺</td>
<td>☺</td>
<td>☺☺☺</td>
<td>☺☺☺☺</td>
<td></td>
</tr>
<tr>
<td>SQL Merge Ops: Insert-Select, Update-Join, Delete-Join</td>
<td><strong>Use</strong> TWB Load to staging table</td>
<td>☺☺</td>
<td>☺</td>
<td>☺☺</td>
<td>☺</td>
<td>☺☺☺</td>
<td>☹</td>
<td></td>
</tr>
</tbody>
</table>

*Use Statistics Wizard*
Load Utilities - FastLoad, Multiload, TPUMP

• Restrictions
  • Fastload
    • Inserts only
    • Empty Target Table Required
    • Fallback, Permanent Journal are applied after the Fastload is complete
    • Secondary Indexes, Triggers, Join Indexes and Referential Integrity must be applied after the Fastload is complete.
  • Multiload
    • Unique Secondary Indexes, Triggers, Join Indexes and Referential Integrity must be dropped before and recreated after the Multiload.
  • TPUMP
    • No Restrictions!

Best for mixed workloads & real time data availability.


**“Update” Utilities and SQL: Restartability & Accessibility During Load**

<table>
<thead>
<tr>
<th>Update Method</th>
<th>Checkpoint / Restart</th>
<th>Rollback</th>
<th>Permanant Journalling</th>
<th>Locking &amp; Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWB Load (FastLoad)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Exclusive Write</td>
</tr>
<tr>
<td>TWB Update (MultiLoad)</td>
<td>ACQ: per your specification. Apply: Every datablock</td>
<td>No</td>
<td>Yes</td>
<td>ACQ: Table Access Apply: Table Write</td>
</tr>
<tr>
<td>TWB Stream (Tpump)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Row hash Write</td>
</tr>
<tr>
<td>SQL Merge Ops: Insert-Select, Update-Join, Delete-Join</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Table Write</td>
</tr>
</tbody>
</table>

Best for mixed workloads & real time data availability.
Load Utilities - Teradata Warehouse Builder

- Seamless Integration of Extract, Transformation & Load Ops
  - Parallelizes Extract, Transform and Load Acquisition Operations for improved performance.
  - Data Streams eliminate intermediate data stores.
    - Data Streams are Teradata Warehouse Builder’s merge-able, split-able pipes.
    - Data store limitations not an problem.
      E.g: 2GB Maximum UNIX file size.
    - Less work means even better performance.

- Easy Data Extraction
  - Can extract from heterogeneous data sources, e.g. files, relational tables, sources with different schema.

- Compatible with Partner Products for Complex Transformation
  - Feeds the load utilities with parallelism through load operators:
    - Load (aka Fastload)
    - Update (aka Multiload - insert, update and delete)
    - Stream (aka to TPUMP - insert, update and delete)
    - Export (aka Fastexport).
Load Utilities - Teradata Warehouse Builder

Extract

Source
  \- Read Operator
  \- User Transform Operator
  \- Load Operator

Transform

Source
  \- Read Operator
  \- User Transform Operator
  \- Load Operator

Load

Source
  \- Read Operator
  \- User Transform Operator
  \- Load Operator

Teradata DBMS
Teradata Warehouse Builder Acquisition Phase - Maximizing Performance

- Use Teradata Warehouse Builder features to eliminate ETL steps and intermediate data stores.

- Choose level of parallelism to maximize acquisition performance:
  - More parallel feeds to the point of saturating client
  - Fewer parallel feeds to reduce client management overhead
  - Choice of parallelism application dependent (i.e.: complexity of read and transform operators, speed of source media.)
  - Teradata Warehouse Builder eliminates the old ‘bottlenecked on single source feed’ issue, enabling fuller utilization of Teradata during acquisition phases.
Teradata Warehouse Builder Acquisition Phase - Maximizing Performance

- Consider client processing demands & resource availability
  - Client resources are shared by other tasks
    - (e.g. transformation and read operators)
  - Client CPU demands (most to least):
    - TPUMP > MultiLoad > Fastload

- Consider concurrency effects to yield a saturated DBMS.
  - 2-3 concurrent Fastloads and Multiloads will saturate the DBMS in the apply phase.
  - 1+ utility in apply phase and 1+ utility in the acquisition phase mix well.
  - If one TPUMP is not powerful enough to saturate DBMS, use multiple TPUMPs.
Transformation and Cleansing

- **Where to do it?**
  - Consider the impact on load time.
    - Where is all the required data?
      - Move Teradata Data to Client then load to Teradata: export-transform-load
      - Move Client Data to Teradata: load-transform or load to staging-transform-load
    - Teradata side advantage: *Parallelism*
      - Almost all transformations can be done with SQL/Utilities
  - Guideline:
    - Simple Transformations: Transformation pipe to load utility
    - Complex Transformations: Transform on Teradata DBMS
    - When in Doubt: Measure

- **Can Transformations be eliminated?**
  - Evolve source feeds to a compatible format
Maximizing Performance: More on Simple Transformation

• Definition of Input Data
  • Avoid generated NULLIF & concatenated constructs on .FIELD cmds
    • Use SQL NULLIF and Concatenation for parallelism and reduced client CPU usage.
    • Ie: Do this transformation on the DBMS!
  • Use .FILLER wisely
    • Client can “block copy” input to parcels instead of “field-by-field copy” if no .FILLER, no varchar/byte and no generated fields. Consider bytes saved from transferring via .FILLER vs inefficiencies of “field-by-field copy”.
Maximizing Performance - More on Cleansing

- Unclean data pushes inefficiencies into the apply phases. E.g.
  - Duplicate Unique Index Values
  - Duplicate Rows
  - Constraint Violations
- All unclean data is put to load error tables.
- Multiload error processing is handled one row at a time.
  - Economies of scale lost
- TPUMP error processing causes rollbacks.
  - Highest error processing penalty of all load methods.
- Where to clean?
  - Measure to determine best option.
Load Utilities - FastLoad, Multiload, TPUMP

- Potential Load Rates: Fastest to Slowest
  - **Fastload** is Fastest!

- **Multiload** is fast or slow, depending....
  - Multiload can *almost* yield Fastload rates if the conditions are right:
    - Higher percentage of source data to target table/partition data yields higher load rates
  - Multiload can be slower than TPump if the conditions are wrong

- **TPUMP** has the slowest potential rate
  - TPUMP ALWAYS processes 1 row update at a time.
MultiLoad Acquisition

- **Acquisition Functionality:**
  - Receive data from host and send directly to AMPs.
    - For Deletes & Updates, send only required part of the row.
    - For Inserts, send the whole row.
  - Redistribute & Sort data by hash.
- **Performance trend is linear based on bytes to load.**
  - *This DBMS rate assumes client and connectivity are not bottlenecks.*
    - Customers using big AIX, Solaris and HP-UX with single or dual GB Ethernets seldom have such a bottleneck….

<table>
<thead>
<tr>
<th>Delete/Update Estimate:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(# of rows to update/node)</td>
</tr>
<tr>
<td>(80,000 rows/sec/node)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Insert/Upset/Mixed-Action Estimate:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MBytes to download/node)</td>
</tr>
<tr>
<td>(2.9 MBytes/sec/node)</td>
</tr>
</tbody>
</table>

- **Note:** Delete Task has no acquisition phase.
- **Increase estimated time by 1.6X if table is fallback protected.**
MultiLoad Apply Primary Table

- **Functionality:**
  - Apply sorted data a block at a time into the target table.
- **Performance Trend depends on number of rows/DB affected.**
  - Throughput increases as rows/DB increase to a peak rate.*

### Determining number of rows/DB affected (X):

- \[ X = P \times N \]
  - % of target table affected: \( P = \frac{\text{rows to MultiLoad}}{\text{target table rows}} \)
  - Total rows/DB: \( N = \text{TRUNCATE} \left( \frac{S}{\text{rowsize}} \right) \)
  - Typical DB size (S) is 75% of maximum DB size.
    - e.g., If Max DB size is 63.5K, typical is 63.5K\(^*\).75=48K

- **Example:**
  - 100 rows fit into a datablock (N).
  - We are adding 10% more data to the table. (P)
  - We are therefore adding 10% more data to all datablocks. i.e.: \( X = P \times N = 0.10 \times 100 = 10 \text{ rows/DB affected} \)

---

* Processing time accumulates mostly ‘per row’ and ‘per datablock’, not ‘per byte’.
* Datablock size has some impact on throughput rates.
  - Larger datablocks greatly improve the total response time of an update, but not fully by the corresponding increase in hits/datablock it results in. ‘Per datablock’ processing time is larger with larger datablocks than it is with smaller datablocks.
MultiLoad Apply Primary Table

Apply Time = \( (\text{Rows to load/node}) \times (\text{rows/sec/node rate for your hits/db}) \)
**MultiLoad Apply Primary Table**

- **What about Upsert Performance?**
  - Per datablock, MultiLoad first tries to update the row. If that fails, it re-issues the command within the DBMS as an Insert. (The datablock is still only committed once.)
  - Example for estimating Upsert apply time:
    - Upsert of 100,000 rows will result in 50% Inserts, 50% Updates.
    - Apply time $\approx$ time to do 100,000 updates PLUS time to do 50,000 inserts.

  Use the insert and update rates at the hits/db you get with the original 100,000 rows. (not 150,000 or 50,000 rows.)
**MultiLoad on NUPI Tables / Lumpy Inserts**

- All data shown thus far assumes prime index is UPI.
- MultiLoad with highly non-unique NUPI can reduce performance.
  - Multiset reduces this difference to insignificant by eliminating duplicate row checking.
  - NUPI MultiLoad (w/ or wo/ Multiset) with few (100 or less) rows/value performs like UPI MultiLoad.
- But if NUPI improves locality of reference, NUPI MultiLoad can be faster than UPI MultiLoad!
  - **Lumpy** NUPI Inserts can be orders of magnitude faster than UPI Inserts
  - But... Performance rates at X hits/DB as a result of lumpiness do NOT approach performance rates at same X hits/DB when evenly distributed.
MultiLoad on PPI Tables

- All data shown thus far assumes non PPI Tables.
- MultiLoad to a PPI table can greatly improve locality of reference.
- Unlike lumpy NUSI data, performance of this type of locality of reference yields SAME performance benefit as non partitioned / high hits/db situation.

- Consider instead: What’s the hits/db in the target PARTITION?
  - Apply estimated time according to this hits/db.
**MultiLoad Apply NUSI, Fallback, Journal**

- **Fallback**: Double estimated times to load both primary and NUSI tables if fallback.
- **Permanent Journal**: Additional time required to maintain.
- **Functionality of NUSI maintenance**:
  - Generate NUSI change rows while applying primary table rows
  - Sort NUSI change rows
  - Apply NUSI change rows to NUSI tables
MultiLoad - Maximizing Performance

- Go for the highest hits per datablock ratio
  - Do one, not multiple MultiLoads to a single table
  - Do less frequent MultiLoads
  - Load to smaller target tables, or to PPI partitions
    - active vs archive table partitions
  - Reduce your rowsize
    - Multi-Value Compression
  - Use large datablock sizes

Balance your choices against real-time availability goals and impacts on DSS work.
Load Utilities: TPUMP UPI Updates Trend (DBMS Capability)

- DBMS Functionality: Primary Index Access for Inserts, Updates, Deletes on tables as issued from TPump or other

![Graph showing performance trend with line charts for Insert, Update, Delete, Upsert 0%, Upsert 100% Insert, and CPU %]

**Performance Trend**

- Tasks = stmts/request * sessions
- Around 20 tasks/node diminishing returns set in.
- After ~30-40 tasks/node, no further gain.
Load Utilities: TPUMP NUPI Updates Trend (DBMS Capability)

- NUPI cost is minimal.
  - 10% reduced performance at 1000 rows/NUPI vs 1 row/NUPI.
Load Utilities: TPUMP Updates
Fallback and Permanent Journal Costs

- **Fallback:** Reduce throughput by 2X.
  - e.g.: 100 txns/sec No Fallback -> 50 txns/sec w/Fallback
- **Local Journalling (Before, Local After):** Reduce throughput by 1.2X.
  - e.g.: 100 txns/sec No Journalling -> 83 txns/sec w/Local Journal.
- **After Journalling:** Reduce throughput by 1.75X.
  - e.g.: 100 txns/sec No Journalling -> 57 txns/sec w/Local Journal.
Load Utilities: TPUMP Updates
Cost of Index Maintenance

USI Cost
Change row sent to owning AMP.
Additional CPU/USI is 1.0X the CPU path of primary table insert/delete.
E.g: If it takes 100 seconds for the primary inserts/deletes, it will take an additional 100 seconds to update each USI.

NUSI w/ 1 row/value Cost
NUSI change row applied locally.
Additional CPU/NUSI is 0.55X the CPU path of primary table insert/delete.
E.g: If it takes 100 seconds for the primary inserts/deletes, it will take an additional 55 seconds to update each NUSI.

NUSI w/ x rows/value Cost expected to be like NUSI w/1 row/value Cost.

- Join Indexes, triggers, referential integrity, etc. also must be maintained.

*These SI maintenance costs assume index value not updated.*
TPump - Maximizing Performance

- Key to achieving maximum performance is achieving 30+ concurrent tasks/node from the data feed.

- Potential Issues achieving enough tasks/node:
  - Reading Source Media
  - Client processing CPU and availability
  - Connectivity configuration
  - Sessions
  - Pack
  - Errors
  - “Serialize ON”
**TPump - Client Processing**

- **Client CPU is required for the stream load operator plus transformation and read operators.**
  - 1 Client node running stream load operator only can supply about 100 Teradata Nodes to saturation.
    - Assumes PK Updates with no fallback, no indexes, no journal. If indexes, fallback or journal, 1 client node can supply >100 Teradata Nodes.
  - If one Client node is not powerful enough, consider using more.
    - Partition input data to avoid inter-job row-hash collisions.

---

**How much Client CPU do I need to drive my desired TPUMP rate?**
TPump - Maximizing Performance

• TPump Parameters to maximize feed rates: **Sessions**
  • Increase sessions per AMP to saturate client or DBMS
  • Alternatively, increase sessions only to accommodate the maximum load rate desired.
    Sessions, rate and PSF can all be used to limit feed rates.
    PSF is the only way to guarantee a rate.
  • Watch Out! Too many sessions costs
    • Management overhead on client
    • May result in DBMS message congestion
    • May result in row hash lock deadlocks if no serialize.
  • **Try 1 or fewer sessions per AMP.**

![Graph 1: 5200/550MHz TPump: Sessions Impact](image1)

![Graph 2: 5200/450MHz TPump Client: Sessions Impact](image2)
**TPump - Maximizing Performance**

- **TPump Parameters to maximize feed rates: Pack Rate**
  - How many transactions can you fit in a buffer?
    - 2548 Maximum Using Fields (txns * columns) \( \text{z higher limit at V2R5} \)
    - Most likely limit to hit first
  - Maximum Pack is 600. \( \text{z higher limit at V2R5} \)
  - Maximum 1M Buffer holds \( \text{z higher limit at V2R5} \)
  - Request Parcel
  - Data Parcel (Max 65104 bytes less Request Parcel Size)
    - Only issue if data size is larger than 600 bytes/transaction
  - Response Parcel

- **TPUMP will discover your max pack rate for you.**
  - But only let it do this the first time --- discovery is expensive on client CPU!

### Syntax example (2 columns, 2 txns packed)
```
.LAYOUT lay1a;
.FIELD CUSTOMER_ID * INTEGER;
.FIELD CUSTOMER_NAME * CHAR(40);

.DML
INSERT INTO TABLEX (customer_id, customer_name) values (:customer_id,:customer_name);
TPump Generates a macro: databasename.M20000802_173302_30_01_0001.
Request becomes
Using (AA1 int,AA2 char(40),AB1 int,AB2 char(40))
BT;
exec databasename.M20000802_173302_30_01_0001(:AA1,:AA2);
exec databasename.M20000802_173302_30_01_0001(:AB1,:AB2);
```
**TPump - Maximizing Performance**

- **TPump Parameters to maximize feed rates: Pack Rate**
  - Use highest pack rate possible for your transaction.
  - Higher pack reduces client CPU demand in addition to increasing TPump rate.
    - Is number of columns per row preventing high pack? Try this trick: Combine several character fields into a single field then use substr in the SQL...
  - But, higher pack aggravates error processing and partial buffer overhead...

![Graph showing Pack Level Impact on Transactions per Second (Txns/sec/node)](image1)

![Graph showing Pack Level Impact on Client CPU per Transaction (CPU/txn)](image2)

- **5200/550MHz TPump: Pack Level Impact**
- **5200/450MHz Client TPump: Pack Level Impact**

- **Graph Legend**
  - UPI Upsert, 0% Inserts
  - UPI Upsert, 100% Inserts
  - NUPI Insert
**TPump - Maximizing Performance**

- **TPump Parameters to maximize feed rates:**
  - **Minimize Error Processing and Row Hash Deadlocks**
    - Some causes of Error Processing
      - Duplicate Unique Index Values
      - Duplicate Rows
      - ConstraintViolations
    - What happens:
      - Additional DBMS work and client-to-DBMS traffic to **rollback**, resolve, re-send and reprocess **all** transactions in the buffer.
    - Cleanse Input data as much as possible before giving it to TPump

---

**Error Processing example (6 txns packed, error on 3rd, 6th txn)**

Request sent is Insert1/Insert2/Insert3/Insert4/Insert5/Insert6
DBS applies Insert1/Insert2, gets error on 3rd transaction.
DBS rolls back Insert1/Insert2, sends request back to TPump client.
Client re-sends request as Insert1/Insert2/Insert4/Insert5/Insert6
DBS applies Insert1/Insert2/Insert4/Insert5, gets error on 6th txn.
DBS rolls back Insert1/Insert2/Insert4/Insert5, sends request back to TPump client.
Client re-sends request as Insert1/Insert2/Insert4/Insert5
DBS applies Insert1/Insert2/Insert4/Insert5. Request Completed.
Client sends request & DBS applies ErrTab-Insert3
Client sends request & DBS applies ErrTab-Insert6

---

**Work for 6 True Inserts:**

- Total Inserts: 12
- Total Inserts rolled back: 6
- Total requests sent: 5
**TPump - Maximizing Performance**

- **TPump Parameters to maximize feed rates:**
  - **SERIALIZE**

  - Guarantees all input records for a given row will be processed on the same session, in input record order.


  **Cost:**
  - Client CPU Overhead
  - More “partial buffers” for UPSERTs touching same row.
  - Reduction of average pack rate
  - Potential session skewing
  - Have well distributed PI’s.
  - Don’t pre-sort the source data

Only use SERIALIZE ON if conditions dictate...

- When multiple input records might touch the same row **AND** the order of application is important.
- To minimize row hash lock delays when there are non-trivial row hash synonyms
- If using SERIALIZE ON use “-f 10” to keep clumpy NUPI data moving.
**TPump - Maximizing Performance**

- **TPump Parameters to maximize feed rates:**
  
  **Partition Sessions by DML**
  
  - Without this partitioning, pack factor is determined by the lowest common denominator.
    
    - ie: DML with the most columns causes all DMLs to work with a smaller pack factor
  
  - With partitioning, sessions supporting one DML may have a higher pack factor than a session supporting a different DML to achieve more optimal performance.
  
  - Partitioning also improves statement cache hit rates. (Statement cache is per session.)
  
  - Partitioning allows you to specify the number of sessions per DML.

*New at V2R5 / TUF7.0*
Load Techniques: Combining Fastload with SQL

• Basic Loading:
  • FastLoad to staging table ➔ Insert-Select from staging table to target
  • FastLoad to staging table ➔ Update-Join from staging table to target

• Getting the data together for transformations:
  • FastLoad ➔ Transform/Cleanse ➔ Insert-Select

• Data Elimination:
  • Fastload ➔ Delete-Join
  • Delete from tableX where condition;
    • (Just SQL: No Fastload or query from staging table required.)
FastLoad Acquisition

• Acquisition Functionality:
  • Receive data from client, redistribute to correct AMP.
  • Stores data into multiple 508K buffers. (8 * 63.5K)
  • Sorts each buffer individually.
• Performance trend is linear based on bytes to load.
  • This DBMS rate assumes client and connectivity are not bottlenecks.
    • Customers using big AIX, Solaris and HP-UX with single or dual GB Ethernets seldom have such a bottleneck.

\[
\text{Acquisition Time} = \frac{\text{(Mbytes to load/node)}}{5 \text{ Mbytes/sec/node}}
\]
**FastLoad Apply**

- **Apply Functionality:**
  - Each AMP performs 8-way merge-sorts on its buffers.
  - Writes the sorted data to disk.
  - **Performance trend dependent on number of 8-way merge-sorts that must be performed.**
    - Our Merge-level is 3.25
  - Determining your merge-sort level
    - Merge-level = \( \log_8(\text{Kbytes/AMP} / 508\text{KBytes}) \)

![Diagram of FastLoad Apply](image)

Apply Time =

\[(\text{Mbytes to load/node}) \times \text{Your Merge Level}\]

(7.5 Mbytes/sec/node) \[\text{3.25}\]
Throughput reduced by \(\frac{(4-3)}{3} = 33\%\) compared to 3 merge levels.

Throughput reduced by \(\frac{(5-4)}{4} = 25\%\) compared to 4 merge levels.

Throughput reduced by \(\frac{(3-2)}{2} = 50\%\) compared to 2 merge levels.

450,000,000 bytes/amp >> 3.25 Merge Levels >> 7.5MB/sec/node.
Complex Full File Updates - Apply

- Primary table modifications done block at a time.
- Performance trend depends on number of rows/DB affected.
  - Tput increases as rows/db increase to a peak rate.*

\[
\text{Apply Time} = \text{Query time} + \frac{(\text{Rows to update/node})}{(\text{rows/sec/node rate for your hits/db})}
\]

- Processing time accumulates mostly ‘per row’ and ‘per datablock’, not ‘per byte’.
- Datablock size has some impact on throughput rates.
  - Larger datablocks greatly improve the total response time of an update, but not fully by the corresponding increase in hits/datablock it results in. ‘Per datablock’ processing time is larger with larger datablocks than it is with smaller datablocks.

\[
\text{INSERT into tablexxx SELECT * FROM table2xxx WHERE <some rows qualify and PI of both tables is same, (no redistribution)>;}
\]
Complex Full File Updates - Indexes

- For Full File Updates: Usually No Index Management required.
- Full File Inserts and Deletes: Secondary Index modifications done row-at-a-time.
  - It's better to drop and recreate the index unless the number of rows to update are very small, ie:
    \[ \leq 0.1\% \text{ of the table's rows being updated} \]

```
For each SI: \text{SI	extunderscore time} = \frac{(# \text{ of rows to update/node})}{(\text{rows/sec/node rate})}
```

Multiply Time by 2.4x if fallback.
MultiLoad, Tpump, Complex Updates - Maximizing Performance of Index Maintenance

• **Do you really get value from that Secondary Index?**
  • Unless value uniqueness is enough so that queries will choose the index to access the table, don’t create the index.
    • Will the number of rows qualifying from “Where index = value” result in fewer datablocks being accessed than there are datablocks in the primary table?
    • Index Wizard will tell you if your Secondary Index will be used.

• **Consider Sparse JI**
  • Maintenance only required for a small percentage of the rows
  • Remember, Sparse JI’s must be dropped and recreated if you utilize Multiload.

• **Consider drop and recreate secondary indexes**
  • Generally only an option for very infrequent Multiloads or somewhat infrequent Complex Updates
Create Fallback & Indexes

• Create Fallback:
  • Redistribute rows to fallback AMP and generate fallback table.

Create Fallback Time = \( \frac{\text{Mbytes to load/node}}{12 \text{ Mbytes/sec/node}} \)

• Create USI
  • Primary Copy only -- double time if fallback.

Create USI Time = \( \frac{\text{Mbytes to load/node}}{17 \text{ Mbytes/sec/node}} \)

• Create NUSI
  • Primary Copy only -- double time for fallback.

Create NUSI Time = \( \frac{\text{Mbytes to load/node}}{X \text{ Mbytes/sec/node}} \)
A Cross Comparison of Data Load Strategies

- **Real-Time vs Near Real-Time vs Delayed Availability**
  - TPump: Just trickle them in as you get them.
  - Frequent Batch Fastload/Insert-Select or Multiload every few minutes.
    - With and without PPI.
  - Multiload or Fastload/Insert-Select on daily or weekly intervals

- **Scenario:**
  - 3 years x 5 mil rows a day, 100 byte rows
  - No Secondary Indexes
  - No Fallback, No Permanent Journal
  - No Transformation / Cleansing
  - No Client / Connectivity Bottlenecks

*At varying load frequencies, how does performance of TPUMP vs Multiload vs Fastload / Insert-Select compare?*
Much Delayed Availability

1. MultiLoad-Partitions  
2. FastLoad / Ins-Sel -  
3. MultiLoad  
4. FastLoad / Ins-Sel  
5. TPump

Comparison of Load Strategies, Varying Load Frequencies -  
No Indexes

- Multiload
- TPUMP
- Multiload Monthly Partitions
- FLD/Ins-Sel
- FLD/Ins-Sel Monthly Partitions

Load every x days

% of daily resources required

0.00% 2.00% 4.00% 6.00% 8.00% 10.00% 12.00% 14.00% 16.00% 18.00% 20.00% 22.00% 24.00%

0 20 40 60 80 100
Delayed Availability

1. FastLoad / Ins-Sel - Partitions
2. Multiload - Partitions
3. TPump
4. MultiLoad
5. FastLoad / Ins-Sel

Comparison of Load Strategies, Varying Load Frequencies - No Indexes

- Multiload
- TPUMP
- Multiload Monthly Partitions
- FLD/Ins-Sel
- FLD/Ins-Sel Monthly Partitions

% of daily resources required vs. Load every x hours

0.00% 2.00% 4.00% 6.00% 8.00% 10.00% 12.00% 14.00% 16.00% 18.00% 20.00% 22.00% 24.00%

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Real-Time and Near Real-Time Availability

1. TPump
2. FastLoad / Ins-Sel
3. Multiload - Partitions
4. MultiLoad
5. FastLoad / Ins-Sel
- Partitions

Comparison of Load Strategies, Varying Load Frequencies - No Indexes

% of daily resources required

Load every x minutes

- Multiload
- TPUMP
- Multiload Monthly Partitions
- FLD/Ins-Sel
- FLD/Ins-Sel Monthly Partitions
Q&A