hroug

hrvatska udruga oracle korisnika

Scaling To Infinity: A Common Design Flaw

Thursday 18-October 2012

Tim Gorman
www.EvDBT.com

Speaker Qualifications

- Co-author…
  1. "*Oracle8 Data Warehousing*", 1998 John Wiley & Sons
  2. "*Essential Oracle8i Data Warehousing*", 2000 John Wiley & Sons
  3. "*Oracle Insights: Tales of the Oak Table*", 2004 Apress
  4. *"Basic Oracle SQL"* 2009 Apress
  5. "*Expert Oracle Practices: Database Administration with the Oak Table*", 2010 Apress
- 28 years in IT…
  - "C" programmer, sys admin, network admin (1984-1990)
  - Consultant and technical consulting manager at Oracle (1990-1998)
  - Independent consultant (http://www.EvDBT.com) since 1998
  - Rocky Mountain Oracle Users Group (http://www.RMOUG.org) since 1992
  - Oak Table network (http://www.OakTable.net) since 2002
  - Oracle ACE since 2007, Oracle ACE Director since 2012

Agenda

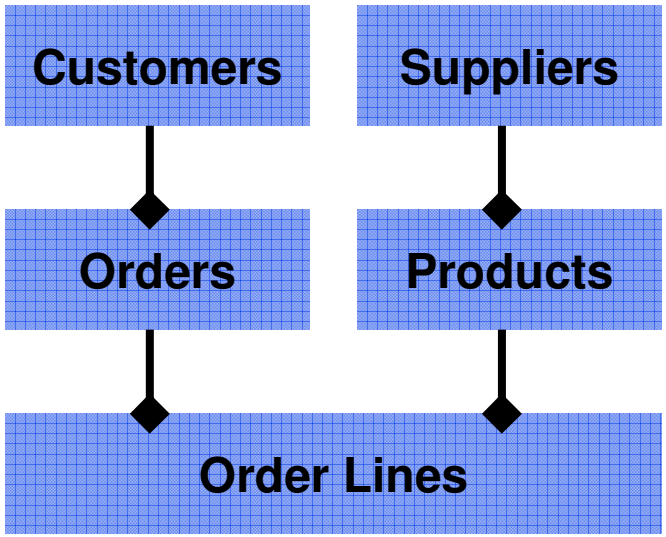
- Brief overview of dimensional data models
- Brief overview of star transformations
- The virtuous cycle and the death spiral
- Portraying time-variant data
- The fatal flaw and how to avoid it

Why Star Schemas?

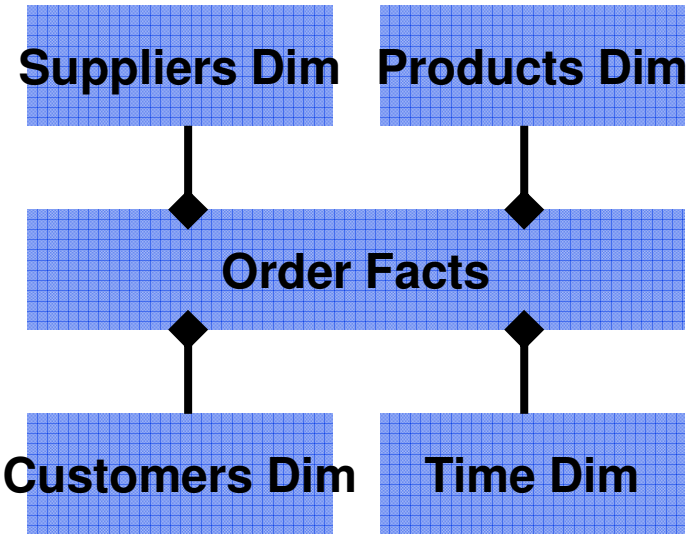
- BI analysts generally just want a big spreadsheet
  - Lots and lots of attribute and measure columns
    - Attributes categorize the data
    - Measures are usually additive and numeric
- *Dimensional data model* is really just that spreadsheet
  - Normalized to recursive depth of one
    - Normalized entities are *dimension tables*
      - Columns are primary key and attribute columns
    - Cells in the spreadsheet are the *fact table*
      - Columns are foreign-keys to dimensions and measures

Why Star Schemas?

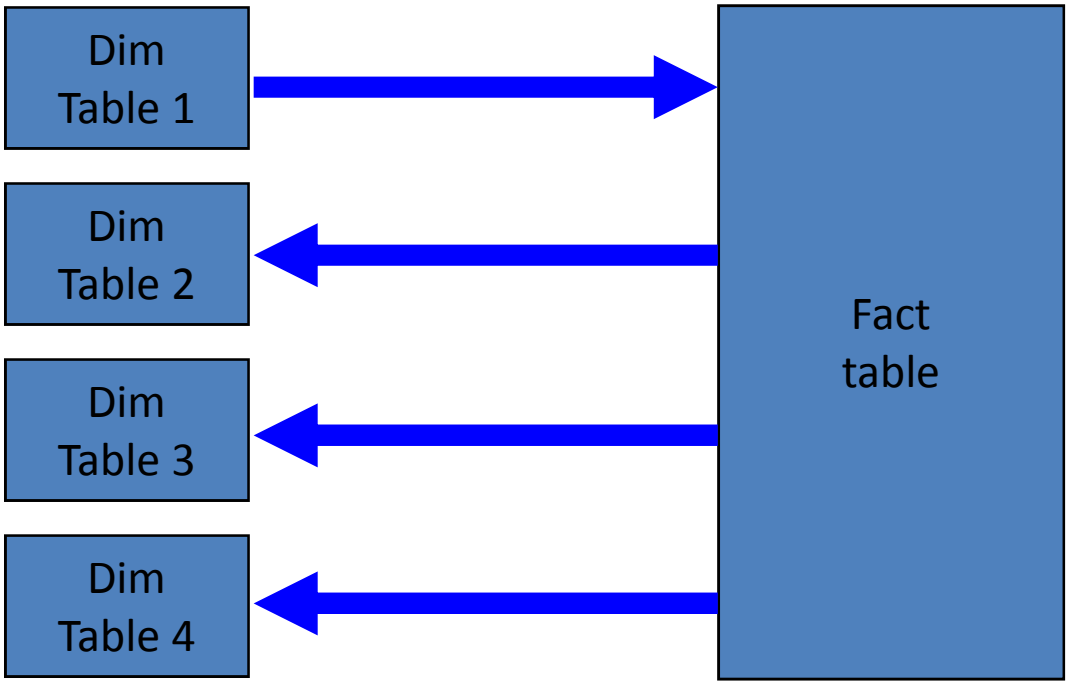
Transactional
Operational
Entity-Relational
Modeling



Dimensional
Modeling



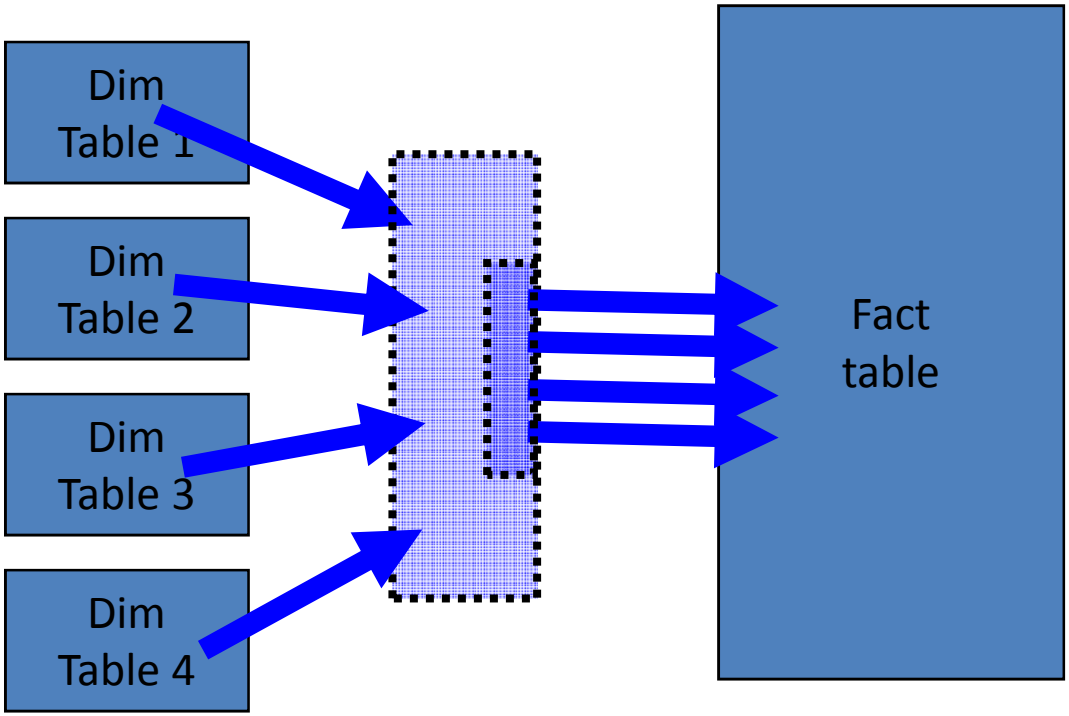
Why Star Transformations?



Star transformation compared to other join methods (NL, SM, HA):

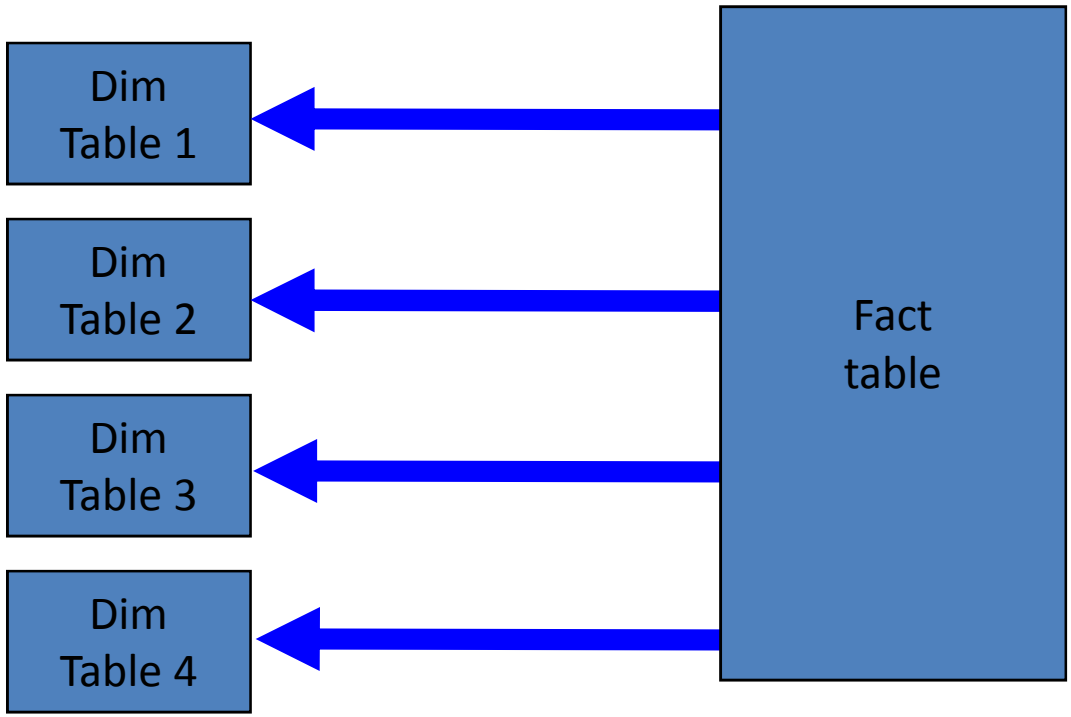
- Filter result set in one of the dimension tables
- Join from that dimension table to the fact along a low-cardinality dimension key
- Join back from fact to other dimensions using dimension PK
  - Filtering rows retrieved along the way
- Wasteful and inefficient

Star Transformation steps



1. Filter WHERE clause on query set in each dimension
2. Merge result set from all dimensions
3. Join to the fact from merged result set, using BITMAP MERGE index scan

Star Transformation steps



4. Join back to dimension tables for items in SELECT clause

Star Transformation reality

- We need star transformations for optimal query performance in the data warehouse.
  - But the mechanism of star transformation requires single-column bitmap indexes on dimension-key columns
  - But bitmap indexes negatively impact data manipulation dramatically
- Does this mean that Oracle database cannot handle large-volume data warehouses?
  - Is star transformation just for demonstration?

Data warehousing reality

- We have to recognize how these features for large data volumes and optimal queries work together
  - Partitioning
  - Direct-path loading
  - Compression
  - Star transformation
  - Bitmap indexes
  - Bitmap-join indexes
  - READ ONLY tablespaces
  - Information lifecycle management
- Because it *really* isn't documented anywhere

The Virtuous Cycle

- Non-volatile time-variant data *implies*…
  - Data warehouses are INSERT only
- Insert-only data warehouses *implies*…
  - Tables and indexes range-partitioned by a DATE column
- Tables range-partitioned by DATE *enables*…
  - Data loading using EXCHANGE PARTITION load technique
  - Partitions organized into time-variant tablespaces
  - Incremental statistics gathering and summarization
- Data loading using EXCHANGE PARTITION *enables*…
  - Direct-path (a.k.a. append) inserts
  - Data purging using DROP/TRUNCATE PARTITION instead of DELETE
  - Bitmap indexes and bitmap-join indexes
  - Elimination of ETL “load window” and 24x7 availability for queries

The Virtuous Cycle

- Direct-path (a.k.a. *append*) inserts *enable*…
  - Load more data, faster, more efficiently
  - Optional NOLOGGING on inserts
  - Basic table compression (9*i*) or HCC (11gR2) for Oracle storage
  - Eliminates contention in Oracle Buffer Cache during data loading
- Optional NOLOGGING inserts *enable*…
  - Option to generate less redo during data loads
  - Optimization of backups
- Table compression enables…
  - Less space consumed for tables and indexes
  - Fewer I/O operations during queries
- Partitions organized into time-variant tablespaces *enable*…
  - READ ONLY tablespaces for older, less-volatile data

The Virtuous Cycle

- READ ONLY tablespaces for older less-volatile data *enables*…
  - Tiered storage
  - Backup efficiencies
- Data purging using DROP/TRUNCATE PARTITION *enables*…
  - Faster more efficient data purging than using DELETE statements
- Bitmap indexes *enable*…
  - Star transformations
- Star transformations *enable*…
  - **Optimal** query-execution plan for dimensional data models
  - Bitmap-join indexes
- Bitmap-join indexes *enable*…
  - **Further optimization** of star transformations

The Death Spiral

- ETL using “conventional-path” INSERT, UPDATE, and DELETE operations
- Conventional-path operations work well in transaction environments
  - High-volume data loads in bulk are problematic
  - High parallelism causes contention in Shared Pool, Buffer Cache
    - Mixing of queries and loads simultaneously on table and indexes
    - Periodic rebuilds/reorgs of tables if deletions occur
    - Full redo and undo generation for all inserts, updates, and deletes
  - Bitmap indexes and bitmap-join indexes
    - Modifying bitmap indexes is slow, SLOW, **SLOW**
    - Unavoidable locking issues in during parallel operations

The Death Spiral

- ETL dominates the workload in the database
  - Queries will consist mainly of “dumps” or extracts to downstream systems
  - Query performance worsens as tables/indexes grow larger
  - Stats gathering takes longer, smaller samples worsen query performance
  - Contention between queries and ETL become evident
  - Uptime impacted as bitmap indexes must be dropped/rebuilt
- Backups consume more and more time and resources
  - Entire database must be backed up regularly
  - Data cannot be “right-sized” to storage options according to IOPS, so storage becomes non-uniform and patchwork, newer less-expensive storage is integrated amongst older high-quality storage, failure points proliferate

So what do we do?

- Fact tables
  - Range-partition by date to aid ETL
    - *Optional*: sub-partition by range, list, or hash to aid queries
- Dimension tables
  - Static dimensions
    - *Optional*: partition or sub-partition to aid queries
  - Slowly-changing dimensions
    - Range-partition by date to aid ETL
      - *Optional*: sub-partition by range, list, or hash to aid queries

Fact tables

- Dimension key columns
  - Foreign keys to dimension tables
    - Combination of all dimension keys represents uniqueness
  - Each key indexed using single-column bitmap indexes
- Degenerate dimension columns
  - Status, type, load date, etc
- Measure columns
  - Additive and numeric for aggregation

Dimension tables

- Primary key columns
  - Static dimensions
    - A single-column, numeric surrogate key
    - For example: geography, products, materials
  - Slowly-changing dimensions
    - A surrogate key plus a timestamp for effective date
    - Examples: people, currencies
  - Enforced with unique index
- Attribute columns
  - Searching, filtering, categorizing/descriptive

Dimension tables

- Primary key columns
  - Static dimensions
    - A single-column, numeric surrogate key
    - For example: geography, products, materials
  - Slowly-changing dimensions
    - A surrogate key plus a timestamp for effective date
    - Examples: people, currencies
  - Enforced with unique index
- Attribute columns
  - Searching, filtering, categorizing/descriptive

Slowly-changing dimensions

- New *versions* of data are added frequently
  - Rows with new version of data are inserted
  - Example: an account dimension

    ACCT_KEY column uniquely identifies a credit-card account…

    …but there might be multiple rows in the dimension table for each ACCT_KEY value
- Slowly-changing dimension must contain timestamp
  - Effective date
- Queries must join fact rows to the appropriate version of dimension
  - Example: it doesn't make sense to associate today's credit card transactions with personal and demographic data from 30 years ago, even for the same person

Slowly-changing dimensions

- But there are situations when the current *point-in-time* data is needed efficiently
  - During data loading
    - Appropriate dimension key value must be assigned to new fact rows
      - Appropriate dimension key is latest row
  - What is the most-efficient way to identify the current *point-in-time* rows?

Identifying current point-in-time

- CURRENT_FLAG column?
  - Set to TRUE when inserted
    - Set to FALSE when newer version inserted
- EXP_DT column?
  - Set to NULL (or 12/31/9999) when inserted
    - Set to SYSDATE when newer version inserted
- EFF_DT column?
  - Set to SYSDATE on insert
    - Never modified when newer version inserted
    - Queries must seek MAX(EFF_DT) for each PERSON_KEY

Identifying current point-in-time

- CURRENT_FLAG column?
  - Set to TRUE when inserted
    - Set to FALSE when newer version inserted
- EXP_DT column?
  - Set to NULL (or 12/31/9999) when inserted
    - Set to SYSDATE when newer version inserted
- EFF_DT column?
  - Set to SYSDATE on insert
    - Never modified when newer version inserted
    - Queries must seek MAX(EFF_DT) for each PERSON_KEY

Here is why…

- Bulk UPDATE processing to set CURRENT_FLAG = FALSE or EXP_DT = SYSDATE initiates the death spiral
  - Conventional SQL operations
  - No partitions can be set READ ONLY
  - No partitions will remain compressed
  - …and so on…

This is the fatal flaw

Here is how to avoid it

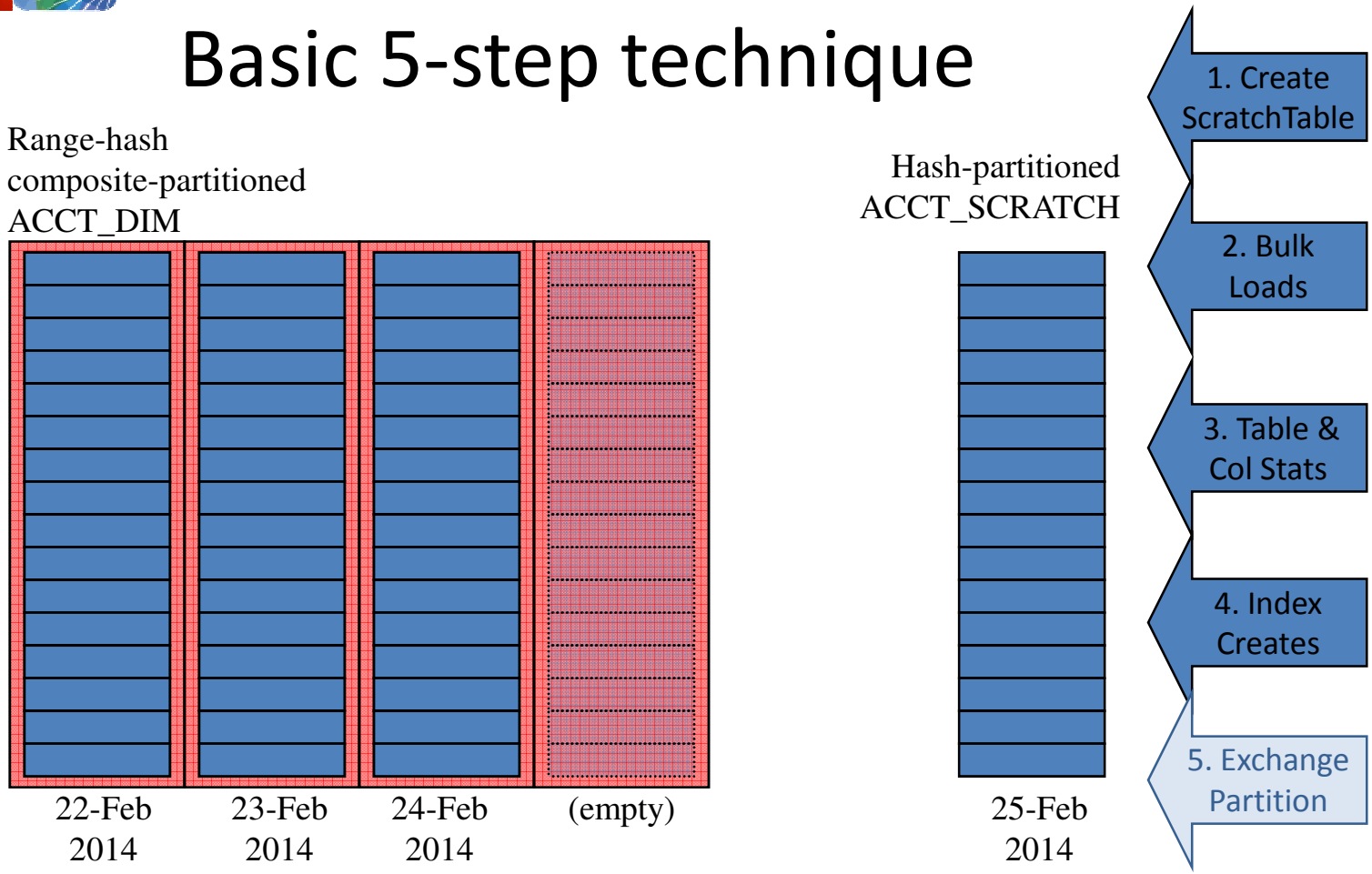
- Each slowly-changing dimension is implemented as two tables
  - One with all versions of data (history)
    - Type-2 (after-change image): time-variant or…
    - Type-3 (before- and after-change): time-variant
  - One with only current point-in-time (non-historical)
    - Type-1, point-in-time, just the latest
- Each load cycle
  - Insert new data into time-variant table first
  - Rebuild point-in-time table
- Anywhere you need to do MERGE logic in bulk, please think of this example

Basic 5-step technique

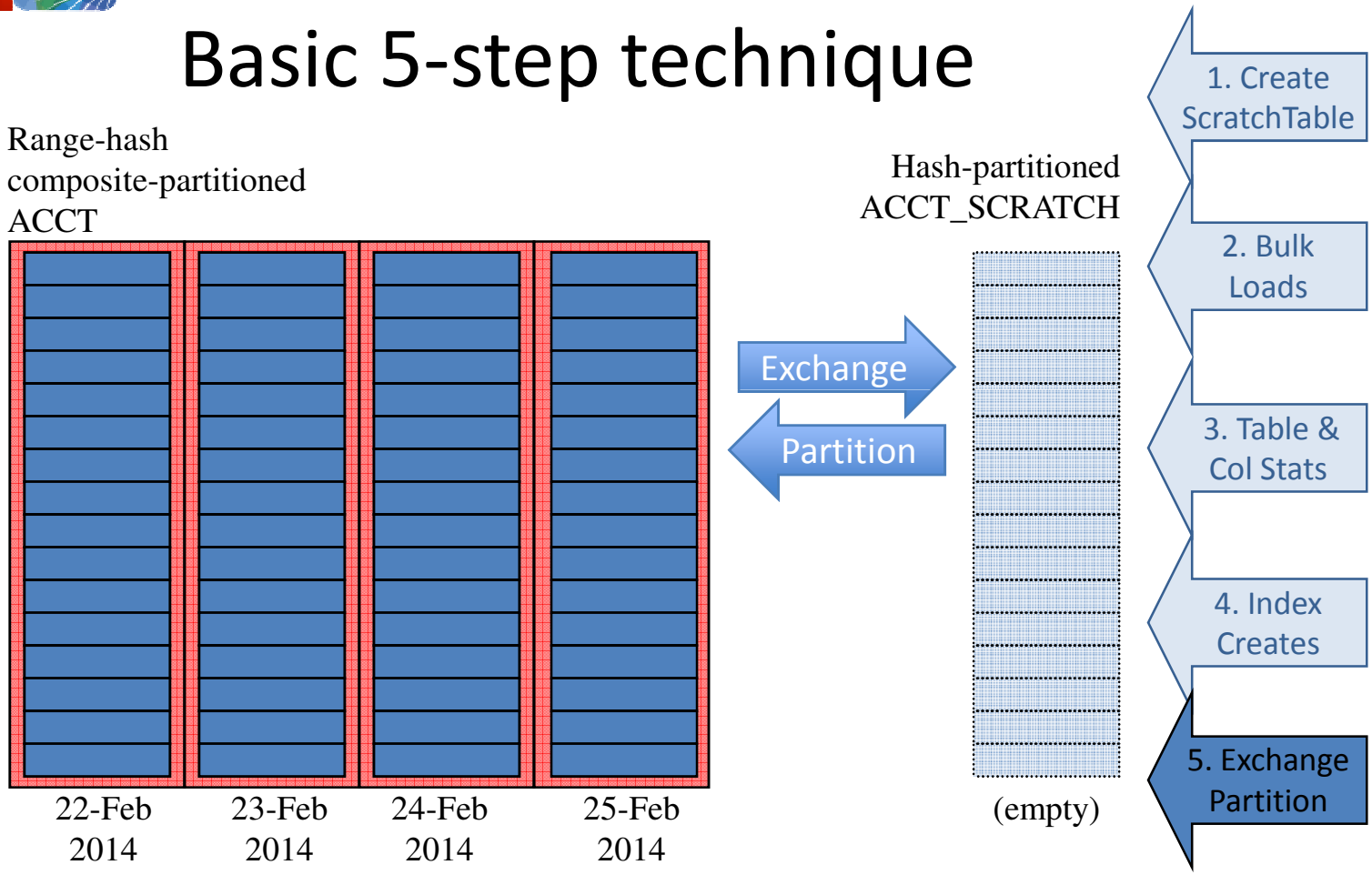
- The basic technique of bulk-loading new data into a “scratch” table, which is then indexed, analyzed, and finally “published” using the EXCHANGE PARTITION operation
  - This should be the default load technique for all large tables in a data warehouse

- Assumptions for this example:
  - A “type 2” time-variant composite-partitioned *slowly-changing dimension* table named ACCT_DIM
    - Range partitioned on DATE column EFF_DT
    - Hash sub-partitioned on NUMBER column ACCT_KEY
  - 25-Feb 2014 data to be loaded into “scratch” table named ACCT_SCRATCH
  - Ultimately data to be published into partition P20140225 on ACCT

Basic 5-step technique



Basic 5-step technique



Basic 5-step technique

1. Create temporary table ACCT_SCRATCH as a hash-partitioned table
   - Must match structure of target partition
2. Perform parallel, append load of data into ACCT_SCRATCH
3. Gather CBO statistics on table ACCT_SCRATCH
   - Only table and columns stats
4. Create indexes on ACCT_SCRATCH matching local indexes on ACCT_DIM
5. alter table ACCT_DIM

   exchange partition P20140225 with table ACCT_SCRATCH

   including indexes without validation update global indexes;

Basic 5-step technique

- It is a good idea to encapsulate this logic inside PL/SQL packaged- or stored-procedures:

```
SQL> exec exchpart.prepare('ACCT_DIM','ACCT_SCRATCH','25-FEB-2014');
SQL> alter session enable parallel dml;
SQL> insert /*+ append parallel(n, 16) */ into acct_scratch n
  3  select /*+ full(x) parallel(x, 16) */ *
  4  from   ext_stage x
  5  where  x.load_date >= '25-FEB-2014'
  6  and    x.load_date < '26-FEB-2014';
SQL> commit;
SQL> exec exchpart.finish('ACCT_DIM','ACCT_SCRATCH');
```

- DDL for EXCHPART package posted at http://www.EvDBT.com/tools.htm#exchpart

Loading the slowly-changing dimension

We could load it this way…

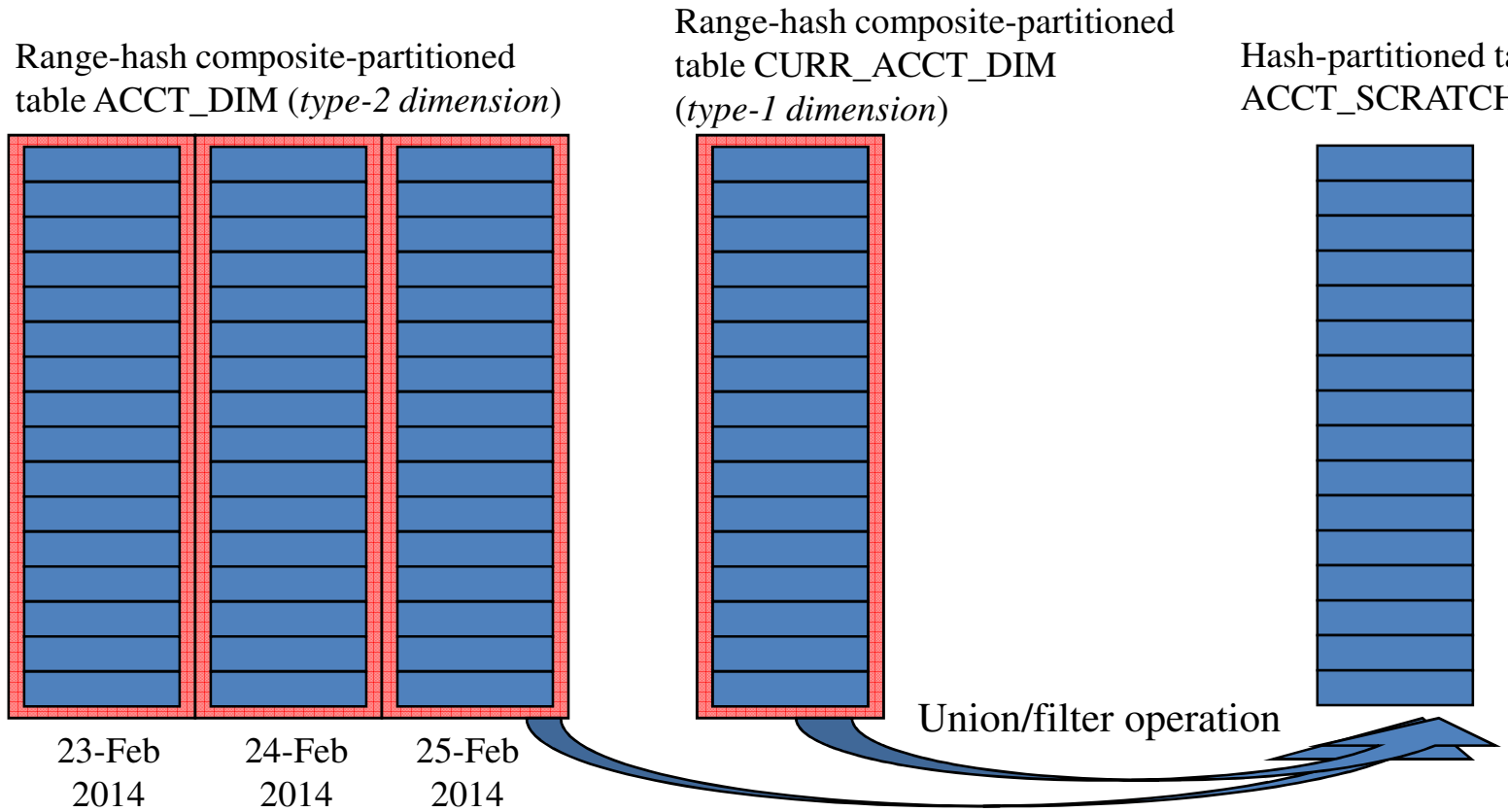
```
merge into curr_acct_dim
using (select * from acct_dim
       where eff_dt >= '25-FEB-2014'
       and   eff_dt <  '26-FEB-2014')
when matched then update set ...
when not matched then insert ...;
```

Loading the slowly-changing dimension

…or we could load it this way instead

1. Create temporary table ACCT_SCRATCH as a hash-partitioned table
2. Perform parallel, append load of data into ACCT_SCRATCH
   - Nested in-line SELECT statements doing UNION, ranking, and filtering
3. Gather CBO statistics on table ACCT_SCRATCH
4. Create indexes on ACCT_SCRATCH matching local indexes on CURR_ACCT_DIM
5. alter table CURR_ACCT_DIM
   exchange partition PDUMMY with table ACCT_SCRATCH
   including indexes without validation;

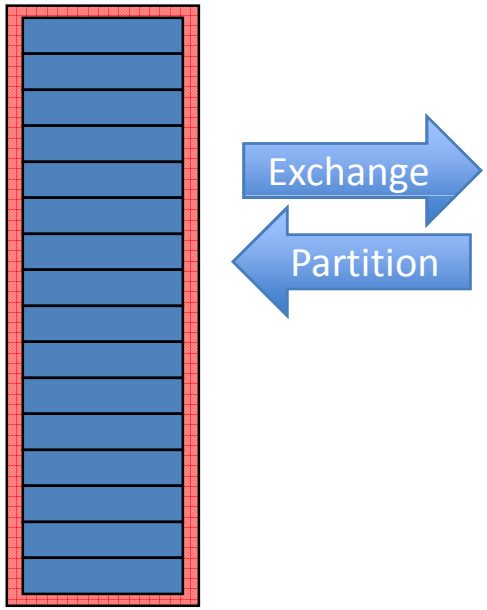
Loading the slowly-changing dimension



Loading the slowly-changing dimension

CURR_ACCT_DIM

- Range-hash composite-partitioned
- Range partition key column = PK column
- Single range partition named PDUMMY
- B*Tree index on PK (local)
- Bitmap indexes (local) on attributes



ACCT_SCRATCH

- Hash partitioned
- Hash parition key column same as CURR_ACCT_DIM
- Indexes created to match local indexes on CURR_ACCT_DIM

Loading the slowly-changing dimension

```
INSERT /*+ append parallel(t,8) */ INTO ACCT_SCRATCH t
SELECT …(list of columns)…
FROM    (SELECT …(list of columns)…,
                ROW_NUMBER() over (PARTITION BY acct_key
                                   ORDER BY eff_dt desc) rn
         FROM   (SELECT    …(list of columns)…
                 FROM      CURR_ACCT_DIM
                 UNION ALL
                 SELECT    …(list of columns)…
                 FROM      ACCT_DIM partition(P20140225)))
WHERE   RN = 1;
```

1. Inner-most query pulls newly-loaded data from ACCT_DIM, union with existing data from CURR_ACCT_DIM
2. Middle query ranks rows within each ACCT_KEY value, sorted by EFF_DT in descending order
3. Outer-most query selects only the latest row for each ACCT_KEY and passes to INSERT
4. INSERT APPEND (direct-path) and parallel, can compress rows, if desired

Loading the slowly-changing dimension

- Assume that…
  - CURR_ACCT_DIM has 15m rows total
  - 1m new rows just loaded into 25-Feb partition of ACCT_DIM
    - 100k (0.1m) rows are new accounts, 900k (0.9m) rows changes to existing accounts
- Then, what will happen is…
  - Inner-most query in SELECT fetches 15m rows from CURR_ACCT_DIM unioned with 1m rows from 25-Feb partition of ACCT_DIM, returning **16m rows** in total
  - Middle query in SELECT ranks rows within each ACCT_KEY by EFF_DT in descending order, returning **16m rows**
  - Outer-most query in SELECT filters to most-recent row for each ACCT_KEY, returning **15.1m** rows
  - Inserts **15.1m** rows into ACCT_SCRATCH

Summary

1. During load cycles, **first** load time-variant type-2 tables…
   - Either using basic 5-step EXCHANGE PARTITION load technique when load cycles match granularity of range partitions…
   - Or using 7-step EXCHANGE PARTITION load technique for "dribble effect" when load cycles do not match granularity of range partitions
2. …**then**, merge newly-loaded data from time-variant tables into point-in-time type-1 tables
   - Using EXCHANGE PARTITION load technique to accomplish merge / up-sert logic faster

Thank You!

Tim's contact info:

- Web: http://www.EvDBT.com
- Email: Tim@EvDBT.com

White Papers: http://www.EvDBT.com/papers.htm

- "Scaling to Infinity" paper by Tim Gorman
- "Supercharging Star Transformations" by Jeff Maresh
- "Managing the Data Lifecycle" by Jeff Maresh

Scripts and Tools: http://www.EvDBT.com/tools.htm

- "exchpart.sql" package