VERTICA

The Shortest Path to Vertica – Best Practices for Data Warehouse Migration and ETL

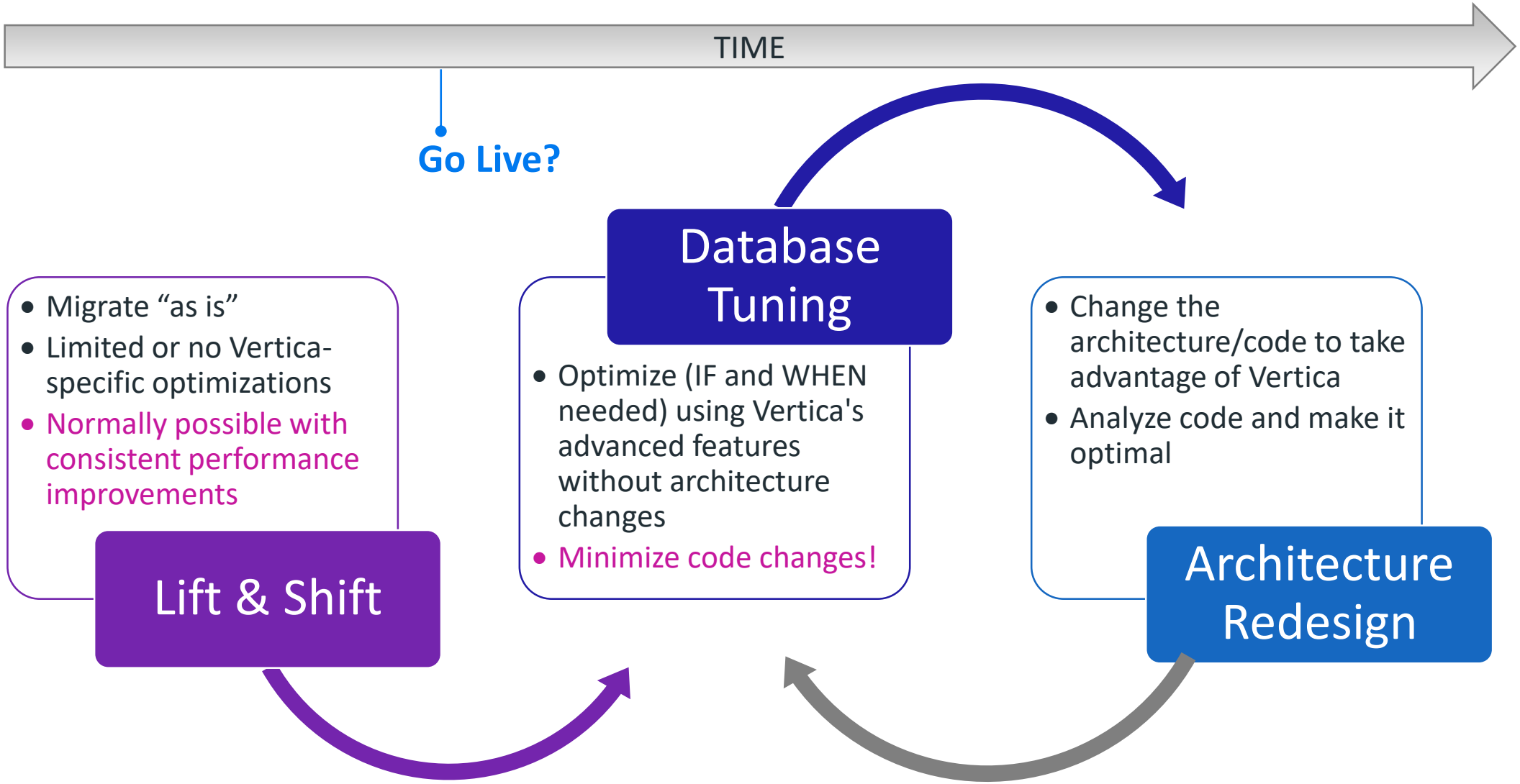
M. Felici, M. Gessner

MICRO FOCUS

Agenda

- **Part 1 – Data Warehouse Migration**
  - Big Bang or Piece By Piece?
  - DDL Migration
  - PDM (re)Implementation
  - ETL/BI Migration
  - Stored Procedures Assessment and Migration
  - UDF Migration
  - Data Migration
- **Part 2 – Data Warehouse Redesign**
  - Vertica-Oriented Data Warehouse Architecture
  - Common Tips & Tricks
  - Live Aggregate Projections
  - Table Flattening
  - Exploiting Vertica Potential

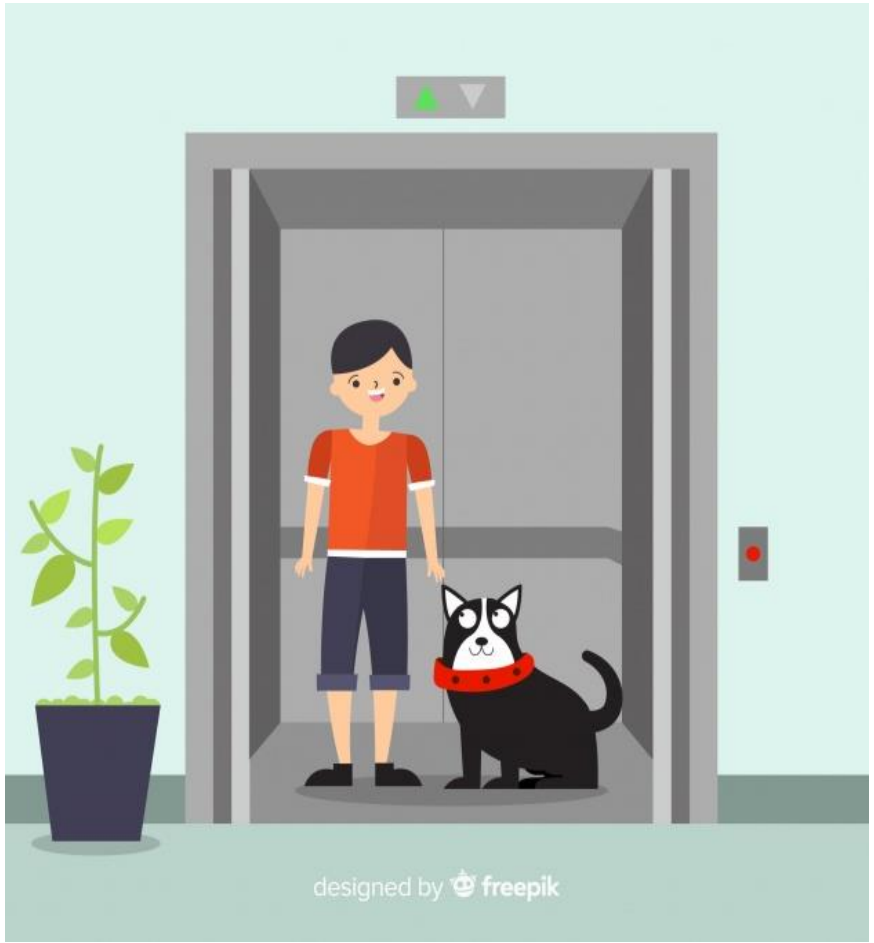
Data Warehouse Migration Process Overview



Instead of breaking chunks out of the wall to accommodate the fork lift for the old St. Bernard who can't walk up stairs any more ...

Learn to use – and trust – the elevator.

You might find altogether new use cases, involving people and their shopping, that you never dreamed of.



VERTICA

Big Bang or Piece by Piece?

- Ideally, picture the new house as your new holiday home
  - Begin to install everything you miss – and everything you like – in your old home
  - Once you have everything you need in the new house, shut down and sell the old one.
- Big Bang only if they are going to retire the platform you are sitting on.
- Otherwise, instead of migrating, transform and free yourself:
  - Identify quick wins
  - Implement and publish them quickly in Vertica
  - Reap the benefits, enjoy the applause
  - Use the gained reputation for further funding of the next iterations
  - Finally, find out no-one is using the old platform any more, and shut it down.
- The following considerations are for the more demanding way of really having to migrate.
  - Big Bang – all in one go, only if you have to
  - Otherwise, whenever you can, stage by Subject Area, User Group, or similar clear divisions

Database Object Migration

- Database object migration is one of the very first steps
- It consists of:
  - Migration of owners and locations of the database objects
  - Object definition extraction from the "old" data warehouse (Tables, Views, etc.)
  - Object definition conversion and deployment to Vertica
- This can be tedious, time consuming, and error prone if executed manually.
- Never type what you can generate.

Database Object Migration

- Users and Roles – export to a script.
  - Old database users' and roles' system tables.
    - If LDAP/ActiveDirectory was used for authentication in the old database – Vertica supports anything within the LDAP standard.
- Catalogs and Schemas
  - While other databases might restrict you by defining a schema as the collection of all objects owned by a given user, Vertica frees you from that.
  - For "old times' sake," it can emulate the owner concept by defaulting to the user name for the schema (before the **public** schema) if a schema with that name exists.
  - "Catalog," in Vertica, is either not needed, or set to the database name.

Database Object Migration

- Tables and their columns:
  - If you are allowed to, it's best to use a tool that translates the data types in the DDL generated:

```
$ odb -d OLD -ddl vertica -i D:%.tpch.%
```

  - If they force you to use what they already have, use that – like SQL*Plus 's DESCRIBE, Teradata's SHOW TABLE command, etc.
  - Each DBMS has a set of tools to extract object definitions to be deployed in another instance of the same DBMS.
- Views usually have their view definition in the old database catalog, which you export to file.
- Synonyms are something that Vertica can emulate in different ways depending on the specific needs.
  - A SELECT * view on the view/table to be referred to by a new name.
  - Vertica's SEARCH PATH

Review the Data Types in the Generated DDL

Never Trust What You Generated

- Dust and clean your stuff before placing it in the new house.
- Don’t do it by hand – use machines for it – we have automated profiling tools to share.
- You will often fix things from previous imperfect migrations!
  - DB2: INTEGER → Oracle NUMBER (NUMBER(38,0))
  - Oracle NUMBER → Vertica NUMERIC (NUMERIC(37,15))
  - Vertica has a real DATE type. Oracle only has a TIMESTAMP(0) as a DATE, and SQL Server makes a mess with datetime types altogether.

```
CREATE TABLE public.foo (
  id numeric(37,15)       -- INTEGER?
, first_name varchar(256)-- shorter?
, last_name varchar(256) -- shorter?
, hire_dt timestamp      -- DATE?
);
```

On Strings

How many bytes does the string '€€€€' contain?

- Vertica stores text in UTF-8 character encoding. If your source database does the same – and has already catered for optimal string lengths, fine.
- Otherwise, strings will be either:
  - *Too long* – and cause memory and other resource waste
    - in queries, especially while grouping or joining by them
      - Well – that's before 9.3.1
    - In any ETL tool or any BI tool connecting to Vertica
- Or:
  - *Too short* – ending up either truncated:
  - or rejected with:

```
COPY: Input record 2 has been rejected (The 17-byte value is too long for type
Varchar(12), column 2 (name))
```

  - At ODBC level:

```
[Vertica][ODBC] (10170) String data right truncation on data from data source: String
data is too big for the driver's data buffer.
```

More on Strings: What's Their Best Size?

How high is the probability of a 4-char-string in Western Europe to actually contain '€€€€'?

- The most promising approach is to initially dimension strings in multiples of their initial length
  - Factor 2 in case of Latin and similar character repertoires
  - Factor 3 for Greek/Cyrillic and similar character repertoires
  - Factor 4 for all other character repertoires
  - Don't re-dimension strings that you already know to contain ASCII characters only.
- We have a tool that can automatically generate such a DDL
- `$ odb -d OLD -ddl vertica -i U2,4:%.tpch.%`
  - load a representative sample of the source data
  - Profile - using our tools

Projection Design

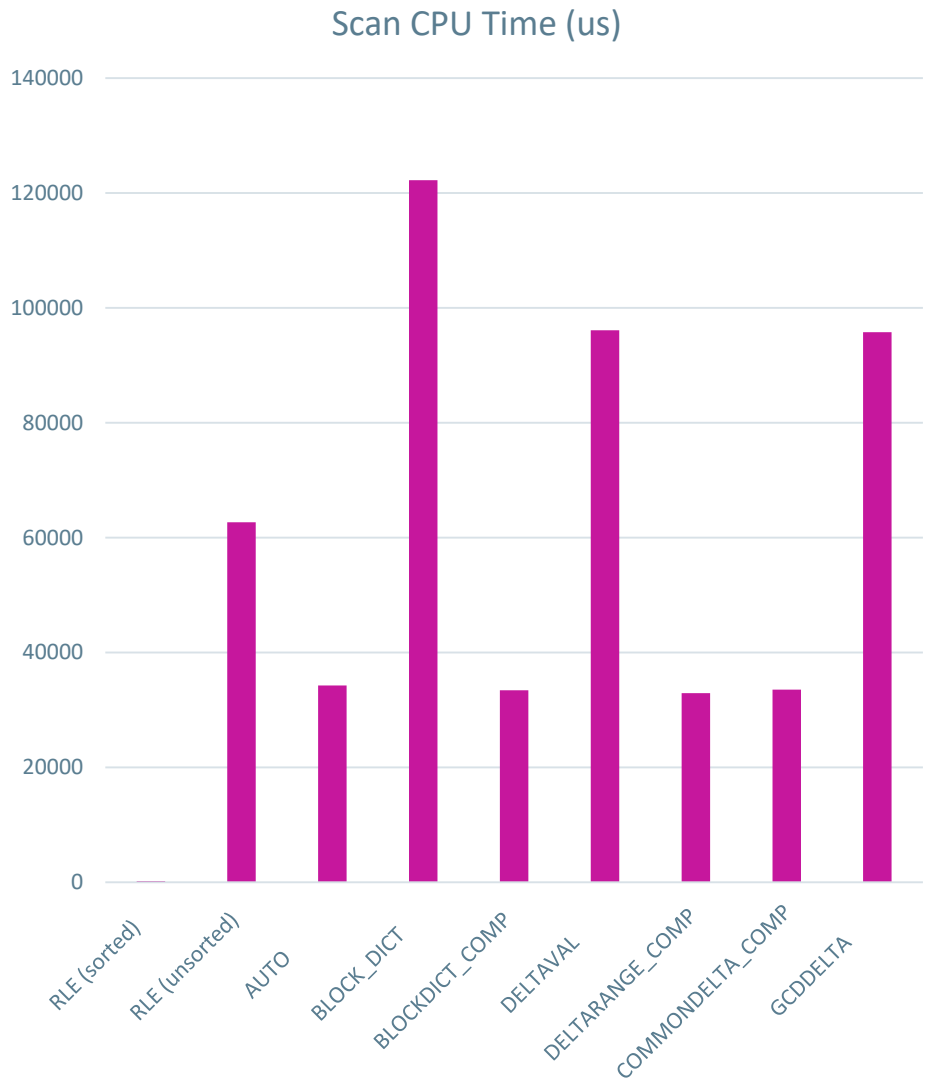
- Remember the rules of how default projections come to exist:
  - Load a representative sample of the data
  - Collect a representative set of already known queries and
  - Run the Vertica Database Designer, for a good set of decisions on sorting, distributing and encoding data.
- Remember that Vertica frees you from having to decide on if and how to distribute a table's data once and for all.
  - You don't need an old MPP database's distribution rules – you can always create a new projection.
- Otherwise, follow the laws of physics:
  - An MPP platform's main bottlenecks are I/O, and the network, and lack of memory.

Projection Design: Encoding Matters!

- This is especially true for Big Data
- Encoding “fine tuning” is a time-consuming process with two (sometime conflicting) objectives:
  - Reduce storage footprint
  - Optimize Query Performance
- ENCODING_TYPE depends on several factors:
  - Data type
  - Cardinality
  - Data sort
  - How data is going to be used
- The Database Designer has a tight-fisted mind:
  - It optimizes everything for the smallest footprint on disk possible.
  - With that, it makes you pay the price for compression – in CPU cycles – while you read and write your data.

CPU Cost Comparison Method – Summary (INTEGER)

Column name	Encoding	Scan CPU Time (us)
srle	RLE (sorted)	209
irle	RLE (unsorted)	62,653
iauto	AUTO	34,242
ibd	BLOCK_DICT	122,264
ibdc	BLOCKDICT_COMP	33,429
idv	DELTAVAL	96,109
idrc	DELTARANGE_COMP	32,934
icdc	COMMONDELTA_COMP	33,540
igcd	GCDDELTA	95,763



14

BI Migrations

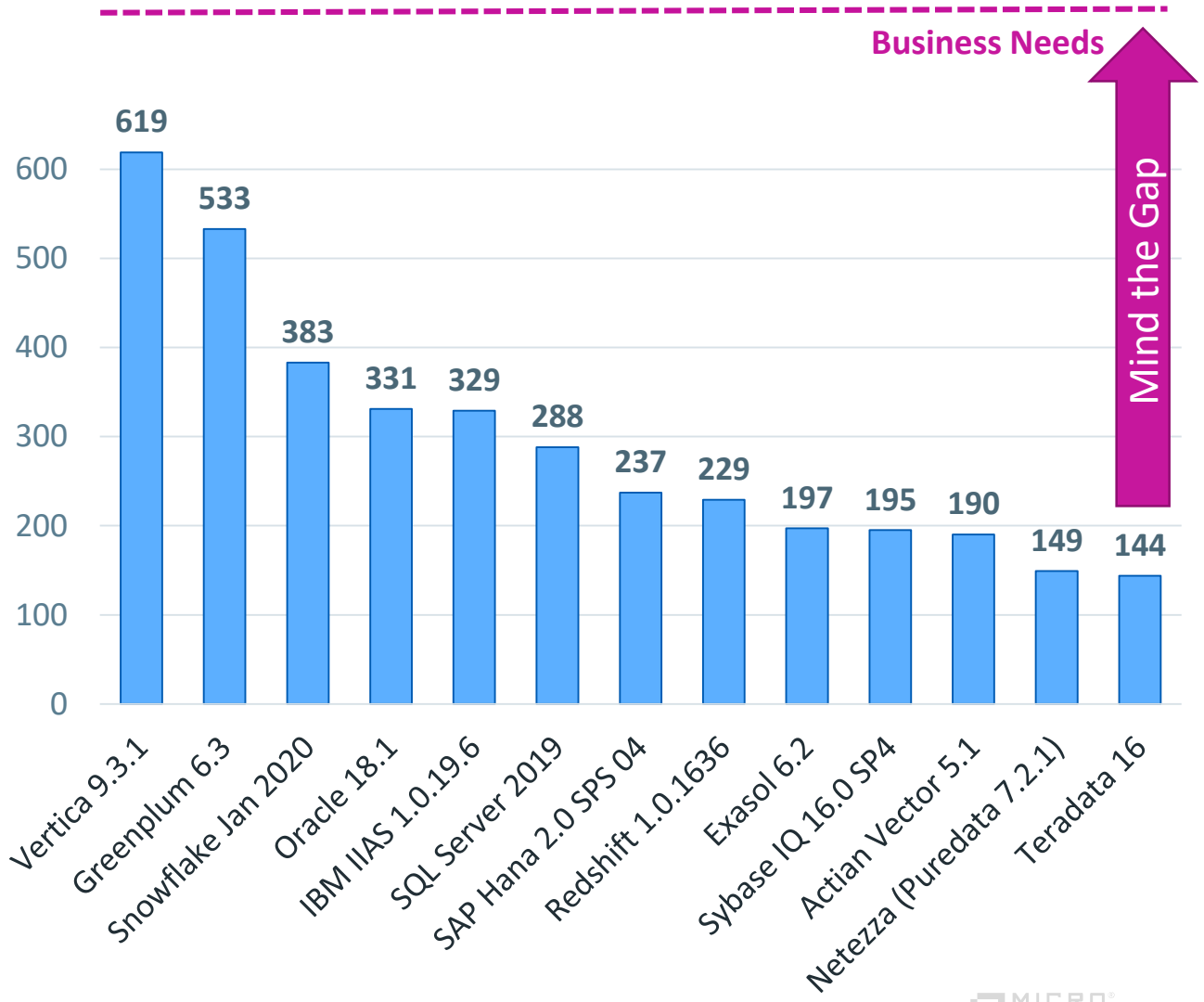
- You can expect the greatest part to be able to be lifted and shifted.
  - Point the database connection objects in the BI metadata from the old database to Vertica, and try everything.
- Many pre-aggregated tables built for popular aggregations can now be Live Aggregate Projections.
- Many BI tools have specialized query objects for facts on one side and dimensions on the other side.
  - In his part, Maurizio will explain Vertica Flattened Tables to you.
  - Don’t build dimensions as SELECT DISTINCT column_list FROM the resulting flattened fact_table; either keep using the dimension tables, or write frequently fired queries manually rather than having them generated.
- Caching frequently used data in the middle tier is counter-productive:
  - The necessary SELECT * queries to fill the cache are not the sweet spot of a columnar database.
  - Vertica aggregates so effectively that this becomes widely superfluous.
- If you worked with MOLAP cubes before, consider replacing that with Vertica Live Aggregate Projections – which will be explained by Maurizio, too.

ETL Migrations

- Many ETL tools have what Informatica Power Center calls “Update Mappings” and “Delete Mappings”. These run an UPDATE/DELETE statement with parameter markers – among which at least one in the WHERE condition, creating a very restrictive filter. Use the biggest possible row sets to process in one SQL call, and: As soon as possible – ideally immediately – change this to:
  - A mass insert into a staging table, which you then use for one single mass DELETE/ UPDATE/MERGE.
  - A mass insert into a staging table partitioned like the target table – and then SWAP_PARTITIONS_BETWEEN_TABLES() the partition keys you find in the staging table.
- Mass inserts from ETL tools – the ones using an INSERT statement with parameter markers – are virtually always inferior to writing to a named pipe, running in parallel with a COPY statement from that named pipe.
  - If the ETL tool supports what PowerCenter calls “Partitioned Sessions”, explore writing in parallel to different CSV files on a drive attached as a local directory to the Vertica cluster, and then run a classic Vertica COPY statement – with apportioned load.

SQL, Procedures, UDF Migration – Mind the Gap!

- SP and UDF **development** is costly.
- Their **maintenance** is often a nightmare.
- So…why do people use them?
- To **fill the Gap** between DB built-in features and their business needs.
- Some database make this gap large.
- Some databases make this gap smaller…



SQL-Based Functions

- The greatest part of SQL-based functions can be migrated with little, automatable, or no modification.
- While it's true that you can't embed a cursor or a SELECT into a Vertica SQL function -
  - Almost always – you don't need it.
  - Vertica has filled the gap that this procedure tried to fill.
  - Vertica does not show the shortcomings this function tried to work around.

Stored Procedures

- Stored Procedures were very often built to overcome shortcomings of the old DBMS when treating mass data:
  - Disturbingly long rollback times in case of the transaction's failure:
    - Our rollback consists in deleting the ROS containers built during the transaction – and that's a very fast operation.
  - Restartability: commit the successful work, so you only need to redo the last bit that failed.
  - Exponentially growing resource consumption for joins and aggregations with the number of rows to be treated:
    - We have no nested joins, for example.
- Rarely, you will have to create a procedure or a function that involves a loop:
  - A custom iterative calculation, for example.
- For this, you will have to plan for development of a Udx in C++, Python, Java, or R.

Data Migration: Start with a Dry Run, then Plan

- These considerations are not necessarily Vertica specific, but worth following in any database migration:
- Migrate a representative subset of real production data: No imagination of any architect or any developer can come up with the same nasty surprises as real data. In the course of that migration:
  - Apply the recommendations as to data types mentioned before: Find your “best” numeric types, date/time types, and string lengths → develop a rule when to use fixed CHARs and when to use VARCHARs.
  - Run an initial Database Designer session.
  - Test different tools for time, if you have a choice, to decide for the migration tool based on your infrastructure.
  - Time the migration processes, so you can predict the migration time
- Based on the migration time estimation you can now make, plan for one – or more – migrations of the delta that will accrue in the old DBMS while you proceed with the data migration.

Most of the DWH Migration Should Be Automatic!

Task	Automated	How	Importance
DDL Migration	YES	odb	crucial
Data Profiling	YES	Marco's profiling tools	game changing
Encoding	YES	DBD	game changing
PDM optimization	YES	DBD	game changing
User Provisioning	DB: Yes; LDAP: mostly	Old platform's tools generate SQL	No objects without owners: crucial
Functions and Procedures	Mostly not needed; Greatest part of needed ones automatic	Text based search-replace; rarely, manual UDx development	Crucial if company's Intellectual Property

Part 2 – Data Warehouse Optimization

A Few, Simple Messages to Start with…

- Optimize only if and when you have to do that
- Keep in mind Vertica peculiarities
- Use alternative approaches to UPDATEs
- Use Live Aggregate Projections to avoid/limit GBYs
- Use Table Flattening to avoid/limit JOINs
- Exploit Vertica potential implementing analysis you did never dream of

Optimize If/When Needed

- In most of the cases, Vertica performs better than the OLD system out of the box.
- If this is ok for you…there's no need to "optimize" anything.
- Set a target before you start optimizing.
- Use DBD to optimize your PDM! Six SQL statements can do better than several "men-months":

```
SQL> SELECT DESIGNER_CREATE_DESIGN('My_Design');
SQL> SELECT DESIGNER_ADD_DESIGN_TABLES('My_Design', 'MySchema.*', true);
SQL> SELECT DESIGNER_ADD_DESIGN_QUERIES('My_Design', '/tmp/all_my_queries.sql');
SQL> SELECT DESIGNER_SET_DESIGN_TYPE('My_Design', 'comprehensive);
SQL> SELECT DESIGNER_SET_OPTIMIZATION_OBJECTIVE('My_Design', 'QUERY');
SQL> SELECT DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY('My_Design',
                                                    '/tmp/optimal_pdm.sql',
                                                    '/tmp/optimal_deploy.sql');
```

Keep in Mind Vertica Peculiarities

- Vertica is tuned for LOAD and QUERY operations.
- Once Vertica writes data to disk… Vertica will **never** change storage container content.
- The fact that Storage Containers are never modified is one of the Vertica's cornerstones.
- This approach ensures several advantages:
  - Minimal locks
  - Multiple COPY can write into the same table in parallel because they operate on different ROS containers.
  - SELECT in READ COMMITTED require no locks and can run concurrently with INSERTs(snapshot isolation).
  - Backup operations are "simple" and "robust“.
  - It allows "historical" queries.
- But, on the other side, DELETEs require a little attention.

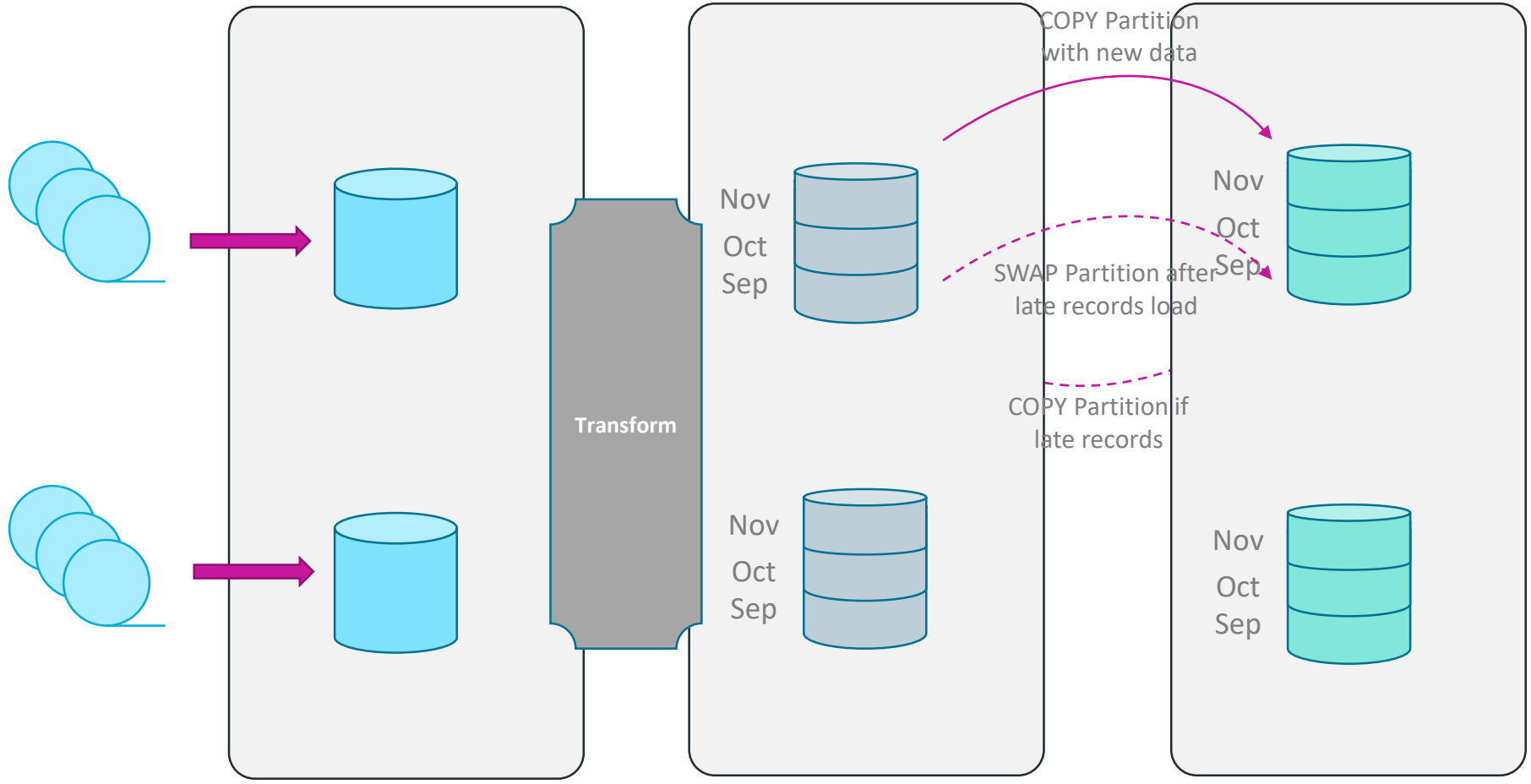
So… What about DELETEs?

- When you DELETE in Vertica, you basically **create** a "Delete Vector" (DVROS or DVWOS) for each storage container containing the data to be deleted.
- The Delete Vector contains the position to the DELETEd rows in the associated storage container and the DELETE EPOCH.
- Then, during a QUERY is executed, Vertica will just ignore rows listed in the Delete Vectors.
- And, is not just about DELETEs:
  - UPDATE in Vertica consists of two operations: DELETE and INSERT
  - MERGE consists of INSERT and/or UPDATE which, in turn, is made of DELETE and INSERT
  - So, basically, tuning for DELETE will let us also tune UPDATE and MERGE ELT cycles.

What Should I Do to Optimize DELETE in My ETL?

- Avoid committing DELETE/UPDATE "too often" to reduce Mergeout activities.
- Remember, we cannot get rid of Delete Vectors more recent than AHM.
- Be sure that all interested projections contain the column in the DELETE predicate.
- Try to segregate UPDATEs/DELETE from QUERY/INSERT workload to reduce lock contention.
- How? For example, using COPY/SWAP Partition.

Best Practices – ELT example with PARTITIONs



VERTICA

MICRO FOCUS

Why Flattened Tables and Live Aggregate Projections?

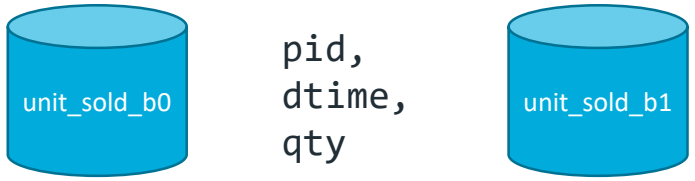
- Compared to *traditional* Data Warehouses, Vertica can:
  - Store orders of magnitude more data
  - Process aggregations orders of magnitude faster: true columnar, encoding, sophisticated optimizations
  - Perform Joins orders of magnitude faster: true columnar, multiple projections per table
- Nevertheless, there are scenarios where data sets are so big and so quickly growing, where we need to boost GBY and Join performance.
- To avoid/reduce expensive queries, Vertica introduces:
  - Live Aggregate Projections to perform Live Aggregations and loading time (and *limit* GBYs)
  - Flattened Tables to combine information from different entities at loading time (and avoid JOINs)

Live Aggregate Projections

- Live Aggregate Projections can pre-aggregate data on the fly using the following built-in functions: **SUM, MIN, MAX, COUNT**
- Let's see how it works…

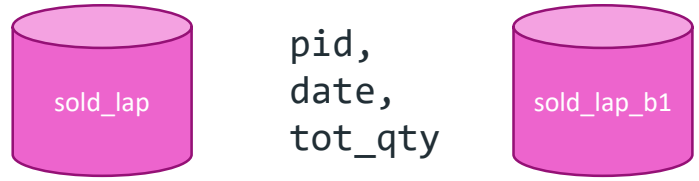
```
DROP TABLE IF EXISTS public.unit_sold CASCADE ;

CREATE TABLE public.unit_sold (
    pid   INTEGER,           -- Product ID
    dtime TIMESTAMP(0),      -- Purshase Timestamp
    qty   INTEGER            -- Unit sold
)
ORDER BY dtime
SEGMENTED BY HASH(pid) ALL NODES KSAFE ;
```

This, after the INSERT, will generate the super-projection **unit_sold_b0** and its "buddy" **unit_sold_b1** if KSAFE=1




```
CREATE PROJECTION public.sold_lap AS
    SELECT
        pid,
        DATE(dtime) AS 'date',
        SUM(qty)    AS tot_qty
FROM public.unit_sold
GROUP BY 1, 2;
```

This, after the INSERT, will AUTOMATICALLY populate **sold_lap (**and its "buddy" **sold_lap_b1)** with **aggregate data**



Live Aggregate Projections – Loading Data

```
INSERT /* +direct */ INTO public.unit_sold
    SELECT 1, '2019-10-09 12:21:32'::TIMESTAMP, 13 UNION ALL
    SELECT 1, '2019-10-09 15:00:11'::TIMESTAMP,  1 UNION ALL
    SELECT 2, '2019-10-10 18:32:01'::TIMESTAMP,  9 UNION ALL
    SELECT 2, '2019-10-09 19:45:21'::TIMESTAMP, 11 UNION ALL
    SELECT 2, '2019-10-09 19:48:29'::TIMESTAMP, 89 UNION ALL
    SELECT 1, '2019-10-10 11:56:21'::TIMESTAMP,  3 UNION ALL
    SELECT 1, '2019-10-10 13:14:56'::TIMESTAMP,  5 UNION ALL
    SELECT 1, '2019-10-10 17:15:23'::TIMESTAMP,  2 UNION ALL
    SELECT 2, '2019-10-10 18:56:21'::TIMESTAMP, 23 ;
COMMIT;
```

Key Concept: Data is loaded into the base (anchor) table. Your ETL processes do NOT have to manage any aggregation.

```
SELECT pid, DATE(dtime), SUM(qty)
FROM public.unit_sold
GROUP BY 1,2;
```

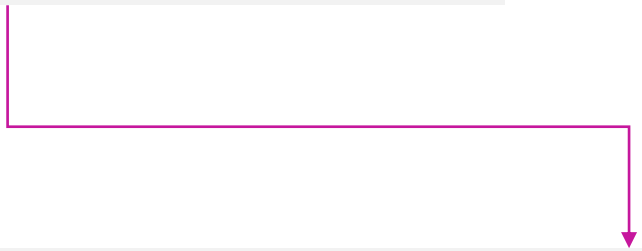
```
SELECT * FROM public.sold_lap;
```

pid	date	tot_qty
1	2019-10-09	14
1	2019-10-10	10
2	2019-10-09	100
2	2019-10-10	32

Key Concept: you do not explicitly run any aggregation. As soon as you load the base table (anchor table), data will be automatically aggregated in the Live Aggregate Projections you have defined.

Live Aggregate Projections – Execution Plan

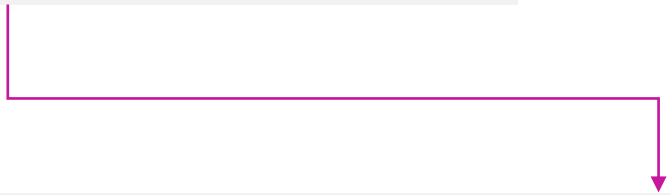
```
SELECT ANALYZE_STATISTICS('public.unit_sold');
EXPLAIN SELECT pid, DATE(dtime), SUM(qty)
FROM public.unit_sold
GROUP BY 1,2;
```



```
Access Path:
+-GROUPBY PIPELINED [Cost: 2, Rows: 4 (NO STATISTICS)] (PATH ID: 1)
|  Aggregates: sum(sold_lap.tot_qty)
|  Group By: sold_lap.pid, sold_lap.date
|  Execute on: All Nodes
| +---> STORAGE ACCESS for public.sold_lap (Rewritten LAP) [Cost: 1, Rows: 4 (NO STATISTICS)] (PATH ID: 2)
| |      Projection: public.sold_lap
| |      Materialize: sold_lap.pid, sold_lap.date, sold_lap.tot_qty
| |      Execute on: All Nodes
```

Live Aggregate Projections – COUNT DISTINCT

```
EXPLAIN SELECT COUNT(DISTINCT pid)
FROM public.unit_sold ;
```



```
+-GROUPBY NOTHING [Cost: 5, Rows: 4 (NO STATISTICS)] (PATH ID: 1)
|  Aggregates: count(DISTINCT sold_lap.pid)
|  Execute on: All Nodes
| +---> GROUPBY PIPELINED (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GROUPS) [Cost: 3, Rows: 4 (NO STATISTICS)]
(PATH ID: 2)
| |      Group By: sold_lap.pid
| |      Execute on: All Nodes
| | +---> STORAGE ACCESS for public.sold_lap (Rewritten LAP) [Cost: 1, Rows: 4 (NO STATISTICS)] (PATH ID: 3)
| | |      Projection: public.sold_lap
| | |      Materialize: sold_lap.pid
| | |      Execute on: All Nodes
```

Live Aggregate Projections Keep Partially Aggregated Data

```
CREATE PROJECTION public.sold_lap
    AS SELECT
        pid,
        DATE(dtime) AS 'date',
        SUM(qty)    AS tot_qty
FROM public.unit_sold
GROUP BY 1, 2;
```

```
INSERT /* +direct */ INTO public.unit_sold
    SELECT 1, '2019-10-09 12:21:32'::TIMESTAMP, 13 UNION ALL
    SELECT 1, '2019-10-09 15:00:11'::TIMESTAMP,  1 UNION ALL
    SELECT 2, '2019-10-10 18:32:01'::TIMESTAMP,  9 UNION ALL
    SELECT 2, '2019-10-09 19:45:21'::TIMESTAMP, 11 ;
COMMIT;
```

```
INSERT /* +direct */ INTO public.unit_sold
    SELECT 2, '2019-10-09 19:48:29'::TIMESTAMP, 89 UNION ALL
    SELECT 1, '2019-10-10 11:56:21'::TIMESTAMP,  3 UNION ALL
    SELECT 1, '2019-10-10 13:14:56'::TIMESTAMP,  5 UNION ALL
    SELECT 1, '2019-10-10 17:15:23'::TIMESTAMP,  2 UNION ALL
    SELECT 2, '2019-10-10 18:56:21'::TIMESTAMP, 23 ;
COMMIT;
```

```
  pid |    date    | tot_qty
------+------------+----------
   1  | 2019-10-09 |       14
   2  | 2019-10-09 |       11
   2  | 2019-10-10 |        9
   1  | 2019-10-10 |       10
   2  | 2019-10-09 |       89
   2  | 2019-10-10 |       23
```

Live Aggregate Projections Keep Partially Aggregated Data

```
SELECT COUNT(*) FROM sold_lap;
COUNT
------
     4
(1 row)
```

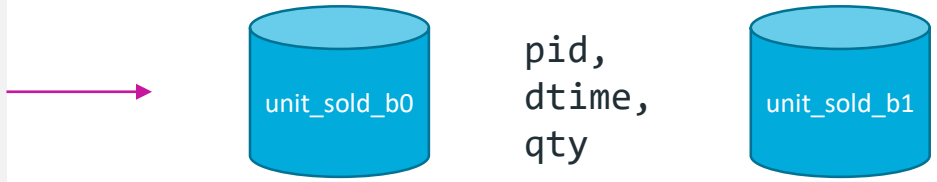
```
SELECT projection_name,
  SUM(total_row_count) AS row_count,
  SUM(deleted_row_count) AS deleted_rows
FROM   storage_containers
WHERE  projection_name LIKE 'sold_lap%'
GROUP BY 1 ;
projection_name | row_count | deleted_rows
----------------+-----------+--------------
sold_lap        |         6 |            0
sold_lap_b1     |         6 |            0
(2 rows)
```

```
EXPLAIN SELECT COUNT(*) FROM sold_lap;
+-GROUPBY NOTHING [Cost: 11, Rows: 1 (NO STATISTICS)] (PATH ID: 1)
|  Aggregates: count(*)
|  Execute on: All Nodes
| +---> STORAGE ACCESS for sold_lap [Cost: 9, Rows: 6 (NO STATISTICS)] (PATH ID: 2)
| |      Projection: public.sold_lap
| |      Materialize: sold_lap.pid
| |      Execute on: All Nodes
```

TOP-k Projections

- Top-K is a LAP variant to return the top (**non-aggregated**) rows using **LIMIT**

```
CREATE TABLE public.unit_sold (
    pid   INTEGER,           -- Product ID
    dtime TIMESTAMP(0),      -- Purshase Timestamp
    qty   INTEGER            -- Unit sold
)
ORDER BY dtime
SEGMENTED BY HASH(pid) ALL NODES KSAFE ;
```

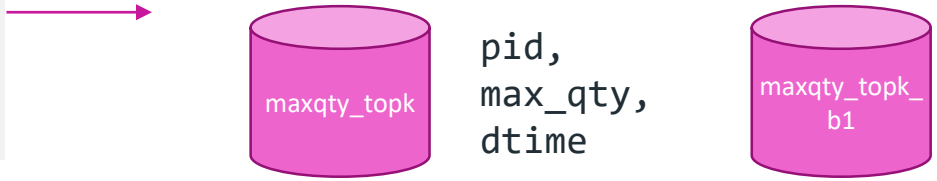



```
CREATE PROJECTION public.lastqty_topk AS
    SELECT
        pid,
        dtime AS last_dtime ,
        qty   AS last_qty
FROM public.unit_sold
LIMIT 1 OVER ( PARTITION BY pid ORDER by dtime DESC ) ;
```

This, after the INSERT, will contain the LAST quantity sold for EACH product and when it was sold.


```
CREATE PROJECTION public.maxqty_topk AS
    SELECT
        pid,
        qty   AS max_qty,
        dtime
FROM public.unit_sold
LIMIT 1 OVER ( PARTITION BY pid ORDER by qty DESC ) ;
```

This, after the INSERT, will contain the MAX quantity sold for EACH product and when it was sold.



TOP-k Projections

```
INSERT /* +direct */ INTO public.unit_sold
    SELECT 1, '2019-10-09 12:21:32'::TIMESTAMP, 13 UNION ALL
    SELECT 1, '2019-10-09 15:00:11'::TIMESTAMP,  1 UNION ALL
    SELECT 2, '2019-10-10 18:32:01'::TIMESTAMP,  9 UNION ALL
    SELECT 2, '2019-10-09 19:45:21'::TIMESTAMP, 11 UNION ALL
    SELECT 2, '2019-10-09 19:48:29'::TIMESTAMP, 89 UNION ALL
    SELECT 1, '2019-10-10 11:56:21'::TIMESTAMP,  3 UNION ALL
    SELECT 1, '2019-10-10 13:14:56'::TIMESTAMP,  5 UNION ALL
    SELECT 1, '2019-10-10 17:15:23'::TIMESTAMP,  2 UNION ALL
    SELECT 2, '2019-10-10 18:56:21'::TIMESTAMP, 23 ;
COMMIT;
```

```
SELECT * FROM public.maxqty_topk ;

 pid | max_qty |        dtime
-----+---------+---------------------
   1 |      13 | 2019-10-09 12:21:32
   2 |      89 | 2019-10-09 19:48:29
(2 rows)
```

```
SELECT * FROM public.lastqty_topk ;

 pid |     last_dtime      | last_qty
-----+---------------------+----------
   1 | 2019-10-10 17:15:23 |        2
   2 | 2019-10-10 18:56:21 |       23
(2 rows)
```

UDTF (Custom) Live Aggregate Projections

- **Live Aggregate Projections** can invoke **user-defined transform functions** (UDTF).
- Currently this is limited to UDTFs coded in C++.
- Two types of UDTFs to implement arbitrarily complex transformations:
  - **PREPASS** Transformations are applied **only during data Ingestion**:

```
CREATE PROJECTION public.my_lap AS
    SELECT id, my_udx(id, data) OVER (PARTITION PREPASS BY id) FROM my_anchor_table;
```

  - **BATCH** Transformations are applied during **pipelined aggregation (Tuple Mover) and SELECT**:

```
CREATE PROJECTION public.my_lap AS
    SELECT id, my_udx(id, data) OVER (PARTITION BATCH BY id)
    AS ( ...output cols...) FROM my_anchor_table;
```

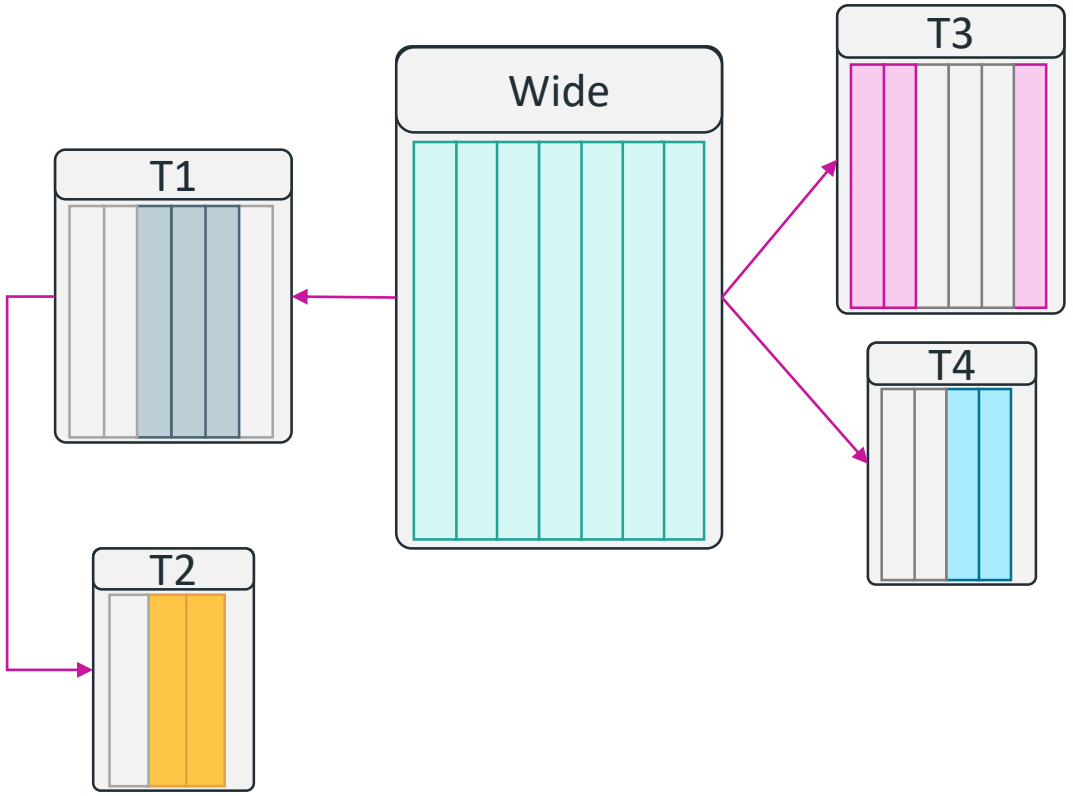
- You can combine BOTH PREPASS and BATCH UDTFs in the same projection definition.

Live Aggregate Projections – Things to Remember…

- Limited to the following **built-in** aggregate functions: SUM, MAX, MIN, COUNT
- Can use **custom built UDTFs functions**
- Can reference only one table
- No UPDATE, DELETE or MERGE on the anchor table **if you are using Vertica < 9.3**
- Cannot be unsegmented (they will be automatically segmented on GBY expression)
- Vertica optimizer automatically pick Live Aggregate Projection if it contains enough data and "enough" aggregation.
- SELECT and GBY expression in the same order with GBY at the beginning

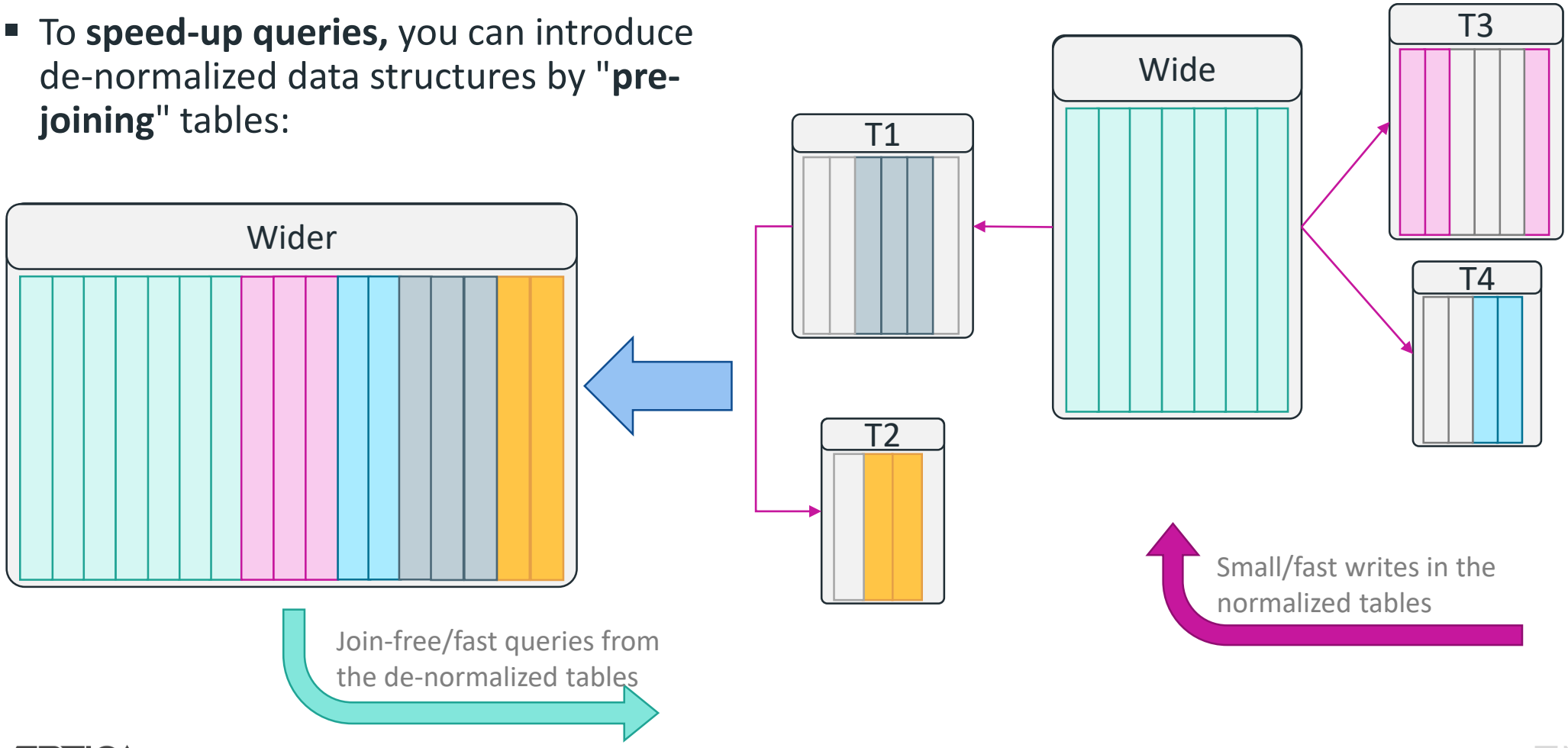
Normalized Schemas

- Normalized Schemas are widely used.
- Main Scope: **reduce data redundancy**
- Optimized for **small/fast writes**
- **JOINs are required** to get information from different tables.
- Sometimes JOINs are difficult to tune.

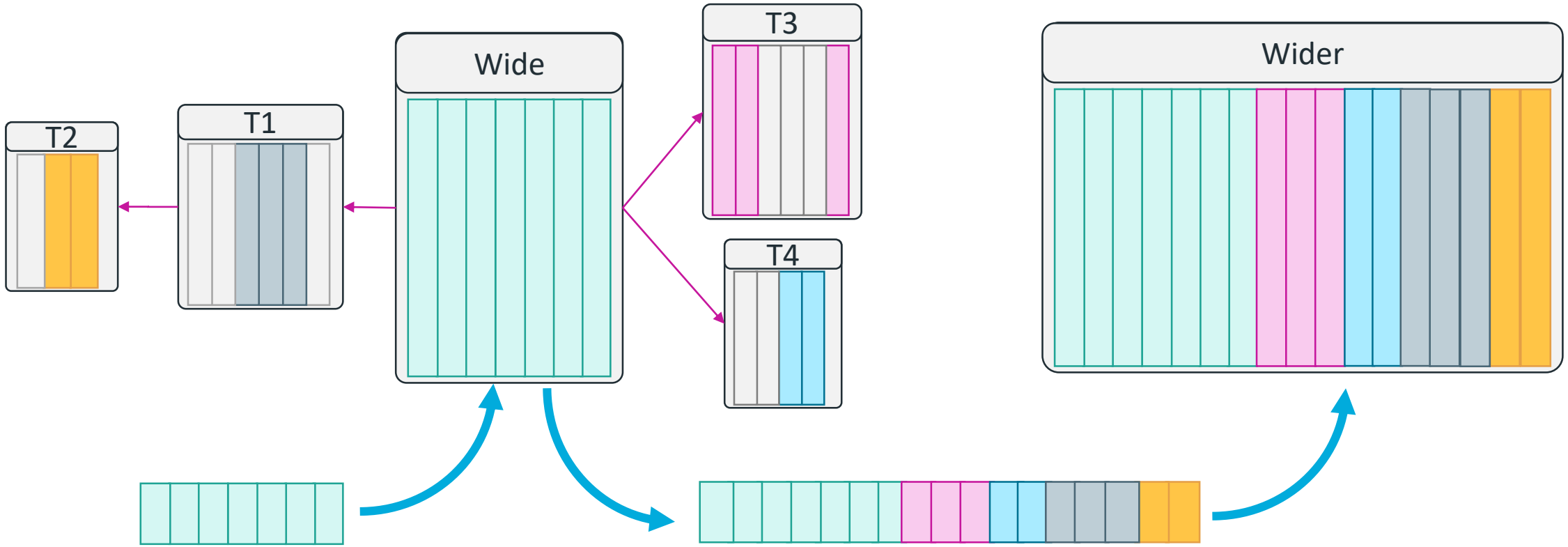


"Manually" De-Normalized Schemas

- To **speed-up queries,** you can introduce de-normalized data structures by "**pre-joining**" tables:



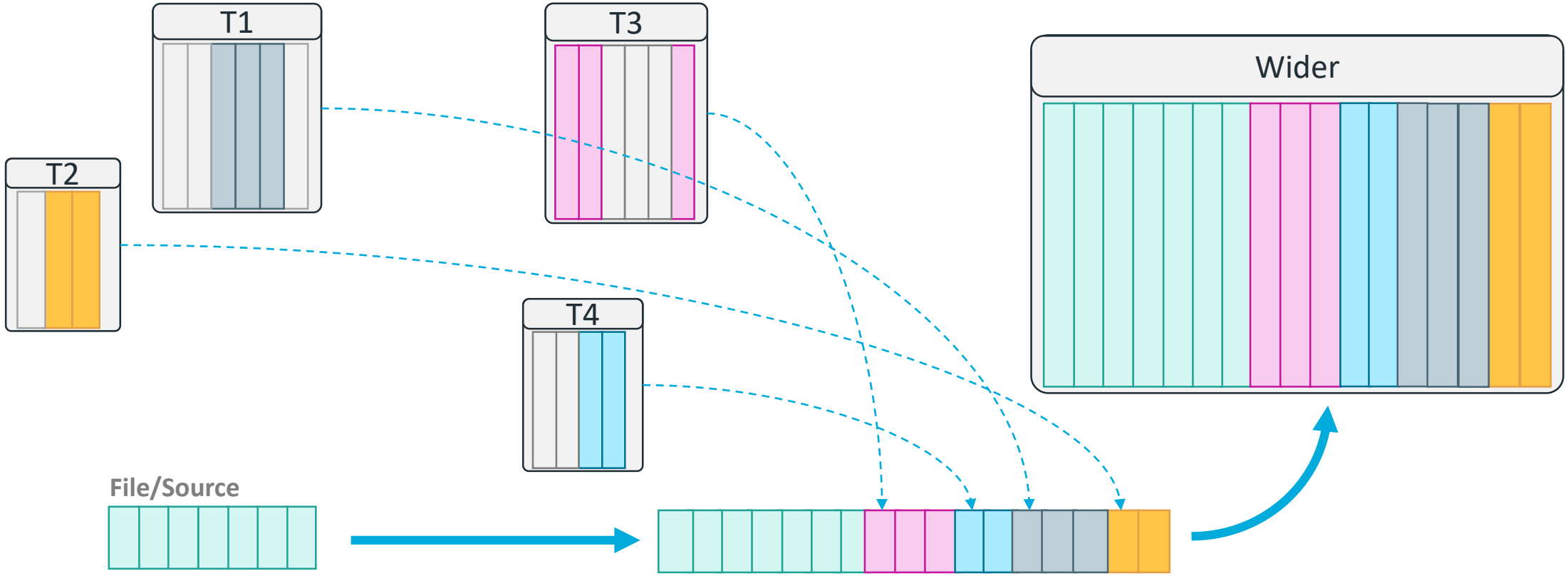
Standard De-Normalization Process



Step 1: Data Loaded in the normalized tables from files/data sources

Step 2: Data is extracted/joined from the normalized schema and loaded into the de-normalized data structures either manually or using ETL procedures.
Problems: This approach is costly, slow and resource intensive.

Vertica De-Normalization Process (Flattened Tables)



Single Step: Data loaded is just loaded into the Wider table and AUTOMATICALLY enriched with data generated on the fly from all the contributing tables.

Advantages: fully automated, fast, avoid data redundancy, robust and… CHEAP!

Vertica Flattened Tables – How They Work

- Columns in Flattened Tables are defined as **query derived from source tables**:

```
CREATE TABLE flattenedTab (
    ...
    dimVal1  INTEGER DEFAULT    ( SELECT val FROM dim WHERE (f_dkey = d_key) )
                     SET USING  ( SELECT val FROM dim WHERE (f_dkey = d_key) )
    ...
);
```

- **Added** and **dropped** as needed, at any point in time
- Populated when:
  - Data of other columns loaded (DEFAULT), or
  - The column is added (DEFAULT), or
  - The column is refreshed (SET USING)

Flattened Table Example (DEFAULT) - DDL

Customer Dimension Table

```
DROP TABLE IF EXISTS public.c_dim ;

CREATE TABLE public.c_dim (
    c_id   INTEGER PRIMARY KEY,
    c_name VARCHAR(20),
    c_age  INTEGER,
    c_city VARCHAR(20)
);
INSERT /* +direct */ INTO public.c_dim
    SELECT  1, 'Alice',  25, 'New York' UNION ALL
    SELECT  2, 'Robert', 19, 'Atlanta'  UNION ALL
    SELECT  3, 'Silvie', 32, 'Boston'   UNION ALL
    SELECT  4, 'Daniel', 43, 'Chicago'
;
COMMIT;
```

Order "Fact" Table

```
DROP TABLE IF EXISTS public.o_fact ;

CREATE TABLE public.o_fact (
    o_oid  INTEGER PRIMARY KEY,
    o_cid  INTEGER REFERENCES
        public.c_dim(c_id),
    o_name VARCHAR(20) DEFAULT (
        SELECT c_name FROM public.c_dim
        WHERE c_dim.c_id = o_cid ) ,
    o_city VARCHAR(20) DEFAULT (
        SELECT c_city FROM public.c_dim
        WHERE c_dim.c_id = o_cid ) ,
    o_tot  NUMERIC(15,2)
);
```

Flattened Table Example (DEFAULT) – Loading data

```
INSERT /* +direct */ INTO public.o_fact(o_oid, o_cid, o_tot)
    SELECT 1001, 1, 134.00 UNION ALL
    SELECT 1002, 2,  78.25 UNION ALL
    SELECT 1003, 3, 260.00 UNION ALL
    SELECT 1004, 1, 415.99 UNION ALL
    SELECT 1005, 1,  87.13 UNION ALL
    SELECT 1006, 4, 189.56 UNION ALL
    SELECT 1007, 1, 231.00 UNION ALL
    SELECT 1008, 3, 370.00;
COMMIT;
```

```
SELECT * FROM public.o_fact ;
```

```
 o_oid | o_cid | o_name |  o_city  | o_tot
-------+-------+--------+----------+--------
  1006 |     4 | Daniel | Chicago  | 189.56
  1001 |     1 | Alice  | New York | 134.00
  1004 |     1 | Alice  | New York | 415.99
  1005 |     1 | Alice  | New York |  87.13
  1007 |     1 | Alice  | New York | 231.00
  1002 |     2 | Robert | Atlanta  |  78.25
  1003 |     3 | Silvie | Boston   | 260.00
  1008 |     3 | Silvie | Boston   | 370.00
```

Key Concept: you do not explicitly run any JOIN. As soon as you load the base table (anchor table) data will be automatically loaded for the "flatten" columns.

Flattened Table Example (SET USING) - DDL

Customer Dimension Table

```
DROP TABLE IF EXISTS public.c_dim ;

CREATE TABLE public.c_dim (
    c_id   INTEGER PRIMARY KEY,
    c_name VARCHAR(20),
    c_age  INTEGER,
    c_city VARCHAR(20)
);
INSERT /* +direct */ INTO public.c_dim
    SELECT  1, 'Alice',  25, 'New York' UNION ALL
    SELECT  2, 'Robert', 19, 'Atlanta'  UNION ALL
    SELECT  3, 'Silvie', 32, 'Boston'   UNION ALL
    SELECT  4, 'Daniel', 43, 'Chicago'
;
COMMIT;
```

Order "Fact" Table

```
DROP TABLE IF EXISTS public.o_fact ;

CREATE TABLE public.o_fact (
    o_oid  INTEGER PRIMARY KEY,
    o_cid  INTEGER REFERENCES
        public.c_dim(c_id),
    o_name VARCHAR(20) SET USING (
        SELECT c_name FROM public.c_dim
        WHERE c_dim.c_id = o_cid ) ,
    o_city VARCHAR(20) SET USING (
        SELECT c_city FROM public.c_dim
        WHERE c_dim.c_id = o_cid ) ,
    o_tot  NUMERIC(15,2)
);
```

Flattened Table Example (SET USING) – Refresh

```
SELECT REFRESH_COLUMNS ('public.o_fact', '');
-- You can refresh specific column names using
-- the second argument of REFRESH_COLUMNS()
-- Empty values for the second argument means ALL columns
```

```
SELECT * FROM public.o_fact ;
```

```
 o_oid | o_cid | o_name |  o_city  | o_tot
-------+-------+--------+----------+--------
  1006 |     4 | Daniel | Chicago  | 189.56
  1001 |     1 | Alice  | New York | 134.00
  1004 |     1 | Alice  | New York | 415.99
  1005 |     1 | Alice  | New York |  87.13
  1007 |     1 | Alice  | New York | 231.00
  1002 |     2 | Robert | Atlanta  |  78.25
  1003 |     3 | Silvie | Boston   | 260.00
  1008 |     3 | Silvie | Boston   | 370.00
```

De-normalized – SET USING - columns are populated through explicit REFRESH_COLUMNS() statements.

Flattened Table – DEFAULT vs. SET USING

	DEFAULT	SET USING
Populate value	Real time (during loading)	When needed (refresh_columns)
Add column	Calculate value	Not calculate value
Sync dimension table change	Need to run UPDATE or DROP/ADD column manually	Call refresh_columns
System table	SELECT column_default FROM columns	SELECT column_set_using FROM columns

- It is possible to use DEFAULT or SET USING for different columns.
- Column can be defined as both DEFAULT and SET USING:

```
CREATE TABLE public.o_fact (
    o_oid  INTEGER PRIMARY KEY,
    o_cid  INTEGER REFERENCES public.c_dim(c_id),
    o_name VARCHAR(20)
        DEFAULT   ( SELECT c_name FROM public.c_dim
                    WHERE c_dim.c_id = o_cid )
        SET USING ( SELECT c_name FROM public.c_dim
                    WHERE c_dim.c_id = o_cid ) ,
    …
);
```


VERTICA
Q&A
MICRO FOCUS

Vertica Academy

Your source for new comprehensive Vertica on-demand training

- Self-Paced Learning
- Online Training
- Knowledge Check
- Certificate of Completion & Certifications

Sign up for free today at **academy.vertica.com**

VERTICA

Vertica Forums

Vertica Engineers are available to answer your questions and moderate discussions right now

For more information visit **forum.vertica.com**