Vertica

Columnar Database Example

Relational vs. Columnar

Row Storage (Traditional)

Symbol	Cmpy Name	Date	Price	Volume
AAPL	Apple Inc.	2014-06-24	90.28	38,988,300
AAPL	Apple Inc.	2014-06-25	90.36	36,852,200
AAPL	Apple Inc.	2014-06-26	90.90	32,595,800
GOOG	Google Inc.	2014-06-24	564.62	2,201,100
GOOG	Google Inc.	2014-06-25	578.65	1,964,000
GOOG	Google Inc.	2014-06-26	576.00	1,726,800

- Storage is organized in rows.
- The DBMS must access entire rows in order to access any column data.

Column Storage (ADS)

Symbol
AAPL
AAPL
AAPL
GOOG
GOOG
GOOG

Cmpy Name
Apple Inc.
Apple Inc.
Apple Inc.
Google Inc.
Google Inc.
Google Inc.

Date
2014-06-24
2014-06-25
2014-06-26
2014-06-24
2014-06-25
2014-06-26

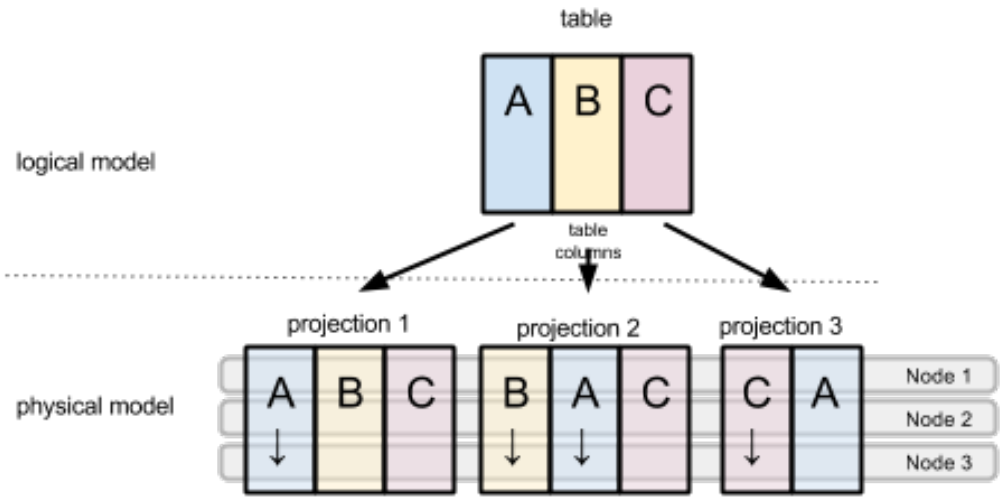
Price
90.28
90.36
90.90
564.62
578.65
576.00

Volume
38,988,300
36,852,200
32,595,800
2,201,100
1,964,000
1,726,800

- Columns are stored independently (or can be grouped).
- When processing a query, Vertica reads only the necessary columns.

Tables and Projections

- A table (logical model) may have many projections (physical layer).
- Projections define how the table columns are physically stored and distributed across cluster nodes.



TABLES AND PROJECTIONS

Projections are Essential to Performance

- Unlike traditional relational databases, Vertica has **no indexes.**
- In place of indexes, projections can be used to optimize query performance.
- As SQL queries are submitted against the logical model, the underlying engine selects the appropriate projection(s).
- Projections can be designed to support different query types on the same logical table

Projections Have Configurable Properties

- Projection parameters that configure the physical data representation include:
  - Columns to be included in the projection
  - Column encoding and compression
    - RLE (Run Length Encoding): sorted repeating values replaced with the value and a number of occurrences
    - AUTO: (default): chooses compression based on data type
    - Delta: several methods for compressing based on changes in values across consecutive rows
    - See HP documentation for others
    - Usage recommendations under Best Practices
  - Sequence of the columns and column data
  - Segmentation (Distribution across cluster nodes)
    - Rows to keep on the same node
    - Projections to be replicated instead of split
    - Unlike tables, there is no default segmentation for Projections!

An Analogy…

- A projection is similar to a materialized view in traditional relational databases.
  - Like a materialized view, projections store result sets on disk instead of re-computing them each time a new query is run.
  - Unlike materialized views, projections are automatically refreshed as data is added or updated.

Superprojections

- A *Superprojection* is a projection that contains all the table columns (matching the logical model).
- The CREATE TABLE command automatically defines a superprojection for the new table.
- Proprietary SQL extensions can be used to configure the parameters of the default superprojection. (details in Best Practices)
- The CREATE PROJECTION command can be used to create additional projections.

CREATE TABLE Example

```
CREATE TABLE  employee_project
        (employee_id    INTEGER,
         project_id     INTEGER,
         start_date     DATE,
         end_date       DATE,
         bill_rate      DECIMAL(7,2) )
        ORDER BY employee_id, start_date;
```

What do you suppose the ORDER BY clause does?

Yes, it defines sequencing for the default superprojection

Simple CREATE PROJECTION Example

```
CREATE PROJECTION employee_project_a
   AS
    SELECT project_id, employee_id, bill_rate
      FROM employee_project
      ORDER BY project_id, employee_id;
```

The syntax is similar to the CREATE VIEW statement.

What types of queries might benefit from this projection?

- Queries searching for all employees on a project

We will see a lot more about projections in Best Practices (the next major topic).

Segmentation

Distributing data across cluster nodes

Segmentation Controls How Data From Tables Is Distributed Across the Nodes in a Cluster

- If segmentation is not specified, default projection data is segmented using a hash of the projection columns.
  - Columns shorter than 8 bytes are taken first
  - No more than 32 columns are used
- Default segmentation is likely to cause joins that span all nodes (broadcast joins), which can be a performance problem at larger data volumes.
  - Default segmentation is not optimal for large joins

Segmentation Options

- Hash Segmentation
  - Specified with `SEGMENTED BY` *`expression`* `ALL NODES` clause within the `CREATE TABLE` or `CREATE PROJECTION` statement
  - The *expression* is expected to return an integer value for each row in the range 0 through 2^{63}.
  - Compute the expression using the `HASH` or `MODULARHASH` function on one or more columns.

(continued on next slide)

Segmentation (Continued)

- Replication
  - Specified with `UNSEGMENTED ALL NODES` clause within the `CREATE TABLE` or `CREATE PROJECTION` statement
  - Recommended for smaller tables that are joined to larger ones
- Range Segmentation is Not Supported
  - This option is prohibited because it ties data to explicitly named nodes (which in turn restricts subsequent reconfiguration of cluster nodes)

Best Practices

Optimize Your Projections

- It's o.k. to start with default projections, BUT…
  - Optimized projections should be created before any live deployment
- Remember that projections are physical copies of columns that must be updated as table data is maintained
  - Projections impede update performance
    - Keep only the ones you really need for queries
    - This is the same tradeoff as you make with indexes in a row-oriented relational database
    - Avoid the urge to optimize for every single query you will run

Analyze Column Ordering and Optimization

- Column Data Ordering: Look for queries with ORDER BY, GROUP BY and JOIN
- Consider these queries:

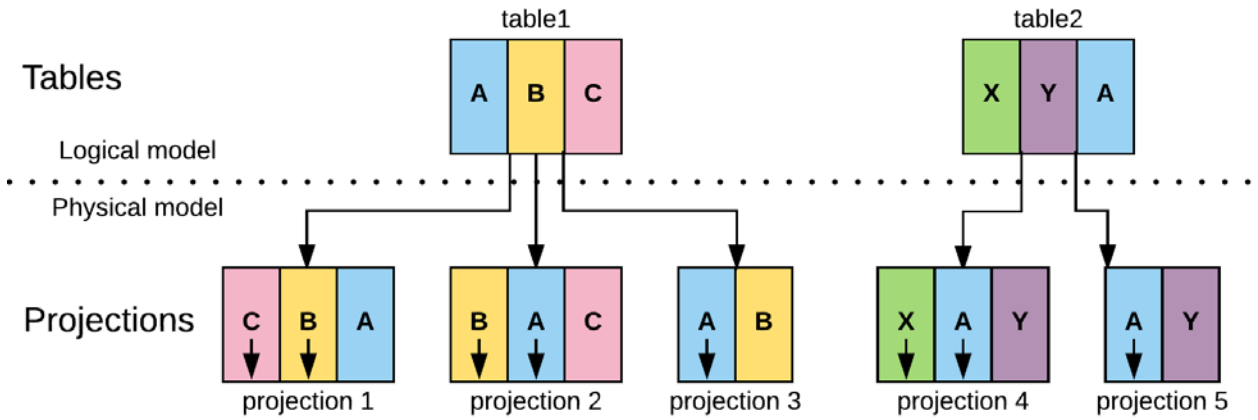
```
SELECT a, b, c  FROM table1
 WHERE c = ‘xxxx’  ORDER BY b;

SELECT b, c, SUM(a) FROM table1
 GROUP BY b, c;

SELECT t1.a, t1.b, t2.y FROM table1 t1
  JOIN table2 t2 ON t1.a = t2.a;
```

Think about the projections you need to optimize each of these

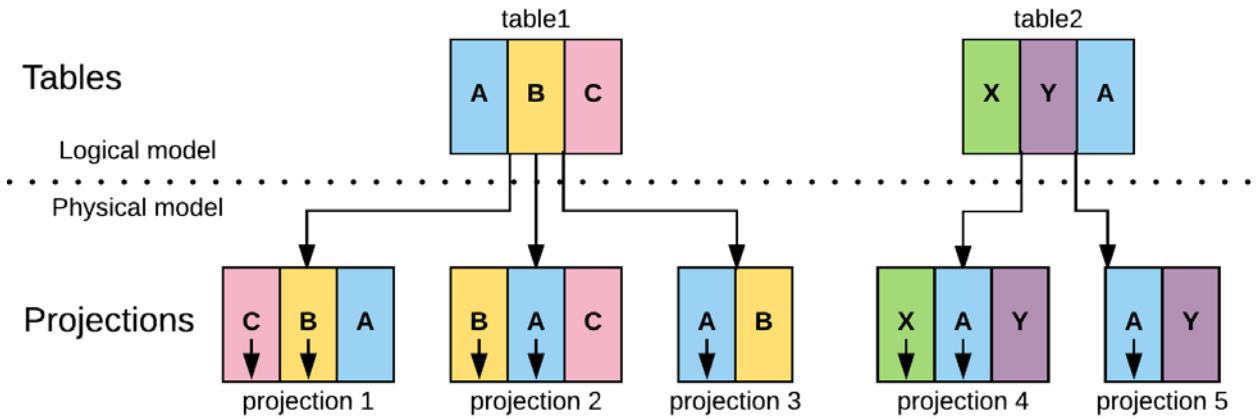
Which Projection is Best for the Query Shown?




```
SELECT b, c, SUM(a) FROM table1
 GROUP BY b, c;
```

- Projection 2 (sorted by B, then A) eliminates the need for the query engine to build a large hash table in order to group rows by B then C – instead it can build a small hash table for each value of B to group just those rows before moving to the next value of B

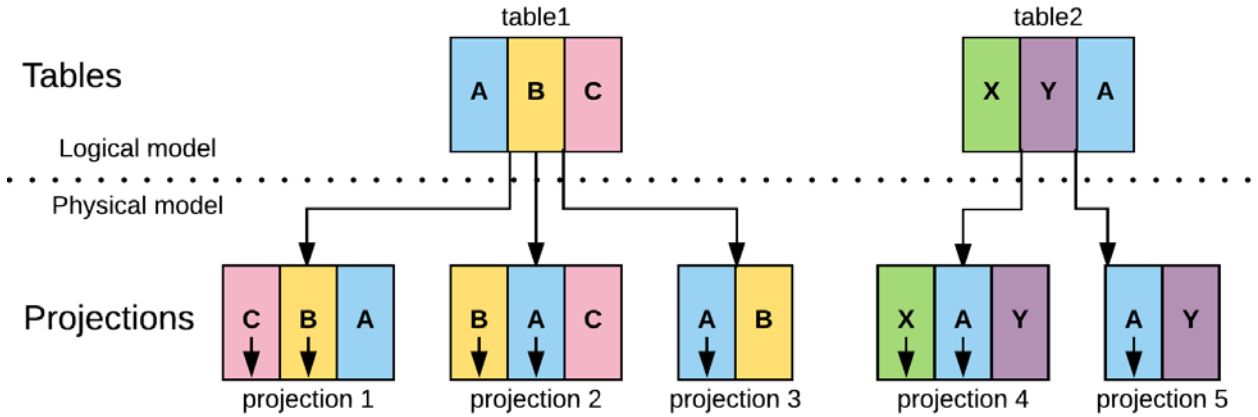
How About This Query?




```
SELECT a, b, c  FROM table1
 WHERE c = ‘xxxx’  ORDER BY b;
```

- Projection 1 (sorted by C, then B) enables quick location of the filtered value for C and returns the result set sorted by B.

Which Projections Are Best for This Join?




```
SELECT t1.a, t1.b, t2.y FROM table1 t1
  JOIN table2 t2 ON t1.a = t2.a;
```

- Projections 3 and 5 allow a merge join of presorted rows instead of loading the smaller table into memory and performing a hash join.
- For larger join operations, a projection sorted on the foreign key column(s) always helps join performance.

Simple Rules for Segmentation

Data Characteristics	Recommended Segmentation
Join of two large tables	Hash segmentation on both tables based on the join key columns (identical segmentation on both tables)
Join of a large table with a small table	Replicate the small table: UNSEGMENTED ALL NODES
Small tables (not involved with joins to very large tables)	Put these tables on a single node: SEGMENTED BY HASH(x) where x is a constant value

- "Small" and "large" are relative terms: in large instances, small tables might contain 1-2 million rows

Help the Optimizer by Collecting Statistics

- The query optimizer uses statistics collected from the data in the tables and/or columns to build an optimal query plan for each query.
  - The `SELECT ANALYZE_STATISTICS` command collects statistics based on a 10% sample of the rows
    - Works fine if data values are uniformly distributed across table rows
  - The `SELECT ANALYZE_HISTOGRAM` command allows you to specify the sample size.

When to Use EXPLAIN to Analyze Queries

- For slow running queries:
  - The EXPLAIN command provides a quick overview of the query plan
  - Examining the query plan may help identify sources of inefficiencies
- Queries can likely always be improved somewhat by explaining the plan and tweaking the physical data structures, but you may spend a lot of time for very little gain
  - In other words, focus on the worst performing queries (be aware of the law of diminishing returns).

EXPLAIN Syntax

- Just place the keyword EXPLAIN in front of your query:

```
EXPLAIN
SELECT product_SKU, transaction_item_id,
       product_quantity, unit_price,
       product_quantity * unit_price AS
extended_price
  FROM wh_transaction_items
 WHERE product_SKU = 'COF087162CQ'
 ORDER BY transaction_item_id;
```

What to Look For in EXPLAIN Results

Access Path:

+-STORAGE ACCESS for wh_transaction_items [Cost: **49**, Rows: 6K] (PATH ID: 2)

Projection: WXc42aBCQWMyGlgufo6gnNvM6CQHCJLv.transaction_items_product_node0003

Materialize: wh_transaction_items.product_SKU, wh_ transaction_items.transaction_item_id, wh_transaction_items.product_quantity, wh_transaction_items.unit_price

Filter: (wh_transaction_items.product_SKU = 'COF087162CQ')

Execute on: Query Initiator

- Cost is a relative number (does not have a unit of measure)
- Watch for RESEGMENT and BROADCAST in joins – they are never good (replicate or segment projections to avoid these)
- Consult HP documentation for more information