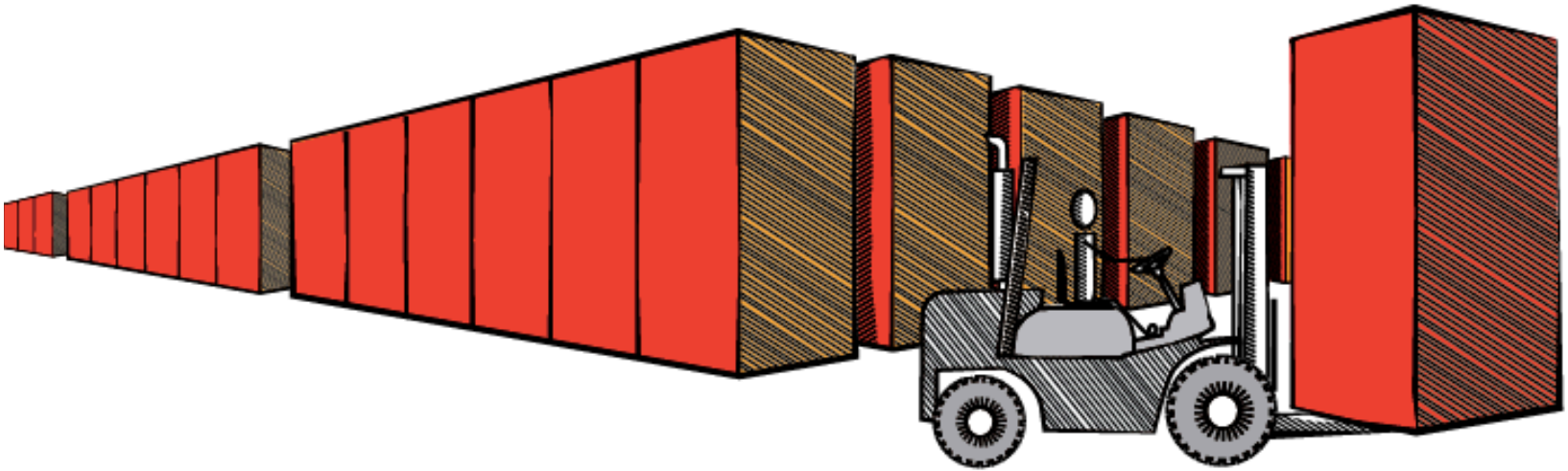


Amazon Redshift Table and Schema Design

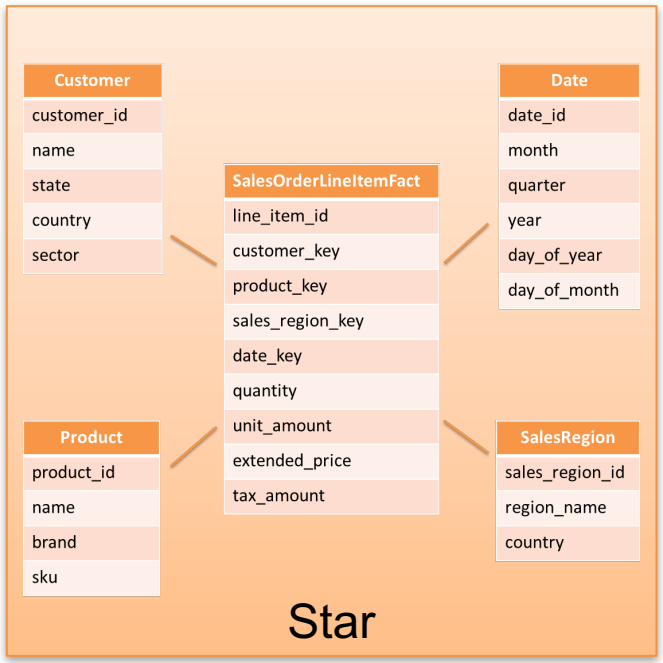
Petabyte-Scale Data Warehousing Service

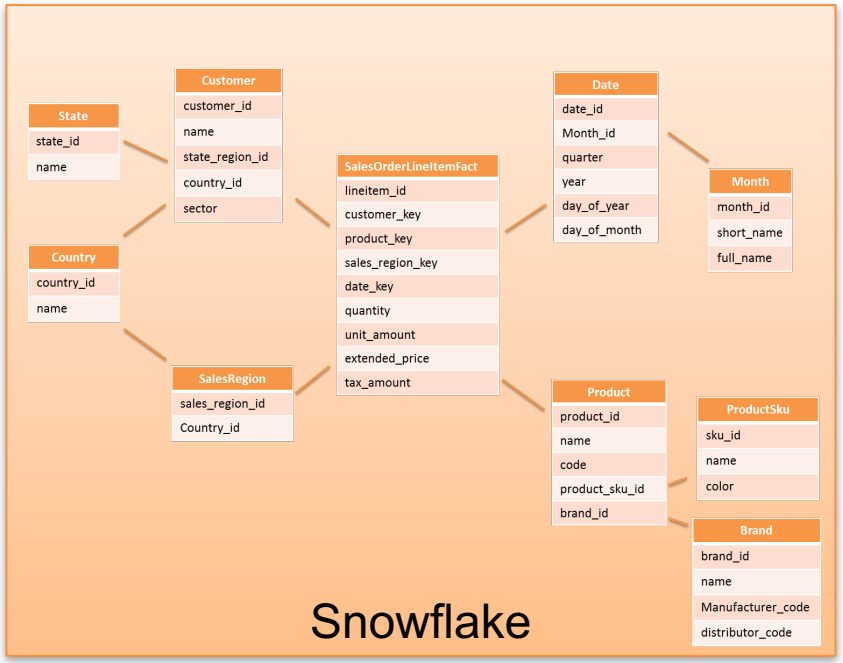
amazon web services

Agenda

- Choosing the right data types
- Distributing and sorting data
- Compression encodings
- Defining constraints
- Optimizing database for Querying
- Lab

Amazon Redshift works with existing data models





amazon web services

Supported Data Types

Data Type	Aliases	Description
SMALLINT	INT2	Signed two-byte integer
INTEGER	INT, INT4	Signed four-byte integer
BIGINT	INT8	Signed eight-byte integer
DECIMAL	NUMERIC	Exact numeric of selectable precision
REAL	FLOAT4	Single precision floating-point number
DOUBLE PRECISION	FLOAT8, FLOAT	Double precision floating-point number
BOOLEAN	BOOL	Logical Boolean (true/false)
CHAR	CHARACTER, NCHAR, BPCHAR	Fixed-length character string
VARCHAR	CHARACTER VARYING, NVARCHAR, TEXT	Variable-length character string with a user-defined limit
DATE		Calendar date (year, month, day)
TIMESTAMP	TIMESTAMP WITHOUT TIME ZONE	Date and time (without time zone)

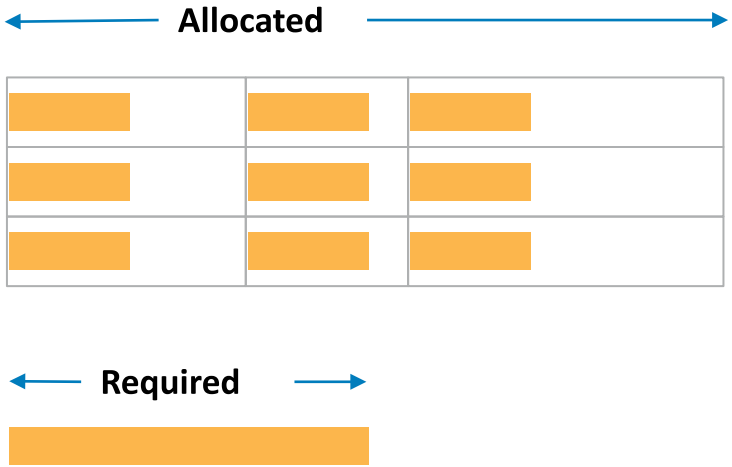
http://docs.aws.amazon.com/redshift/latest/dg/c_Supported_data_types.html

Choosing the Right Data Types

- Redshift performance is about efficient I/O
- Don't make columns wider than necessary, e.g.:
  - Avoid BIGINT for country identifier
  - Avoid CHAR(MAX) for country names
  - Oversizing VARCHAR impact loading and runtime performance
- Use appropriate types
  - Use TIMESTAMP or DATE instead of CHAR
  - Use CHAR instead of VARCHAR when appropriate
- Multibyte Characters
  - VARCHAR data type supports UTF-8 multibyte characters up to a maximum of four bytes
  - The CHAR data type does not support multibyte characters

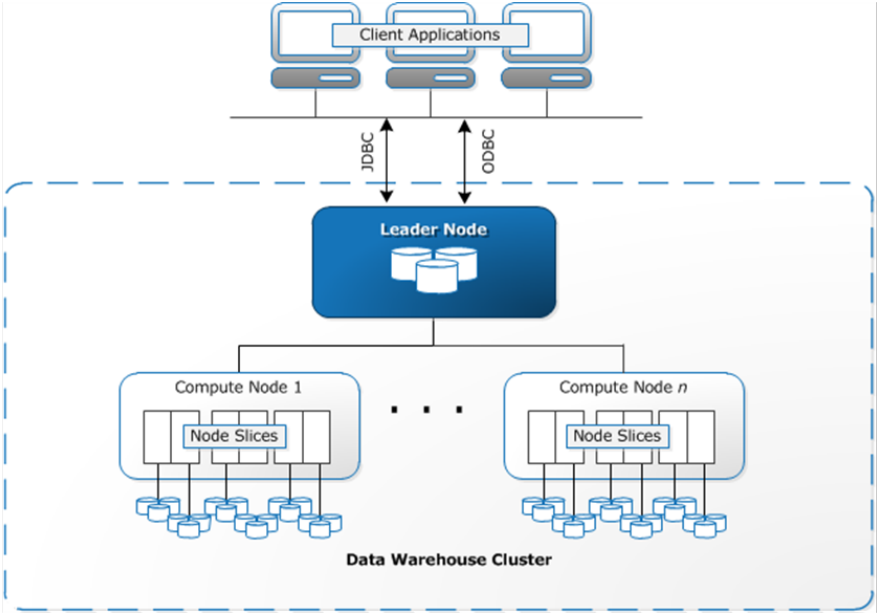
Keep your columns as narrow as possible

- During queries and ingestion, the system allocates buffers based on declared column width
- Wider than needed columns mean memory is wasted
- Fewer rows fit into memory; increased likelihood of queries spilling to disk
- Check SVV_TABLE_INFO(max_varchar)



Architecture and its Table Design Implications

- Redshift is a distributed system:
  - A compute node contains slices (one per core)
  - A slice contains data
- Slices are chosen based on two types of distribution:
  - Round Robin (automated)
  - Based on a distribution key (hash of a defined column)
- Queries run on all slices in parallel: optimal query throughput can be achieved when data is evenly spread across slices

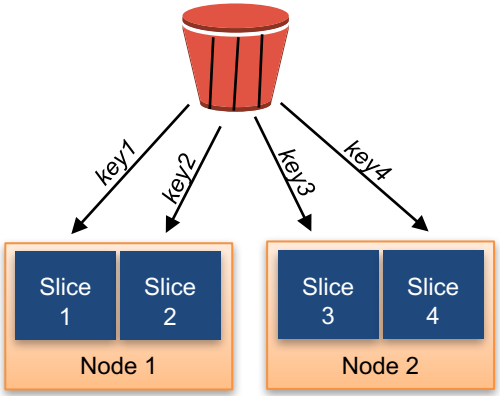


amazon web services

Table Distribution Styles

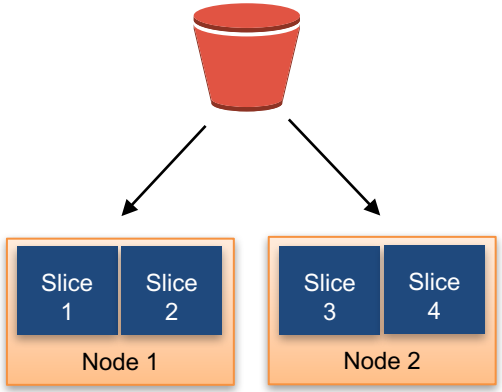
Distribution Key

Same key to same location



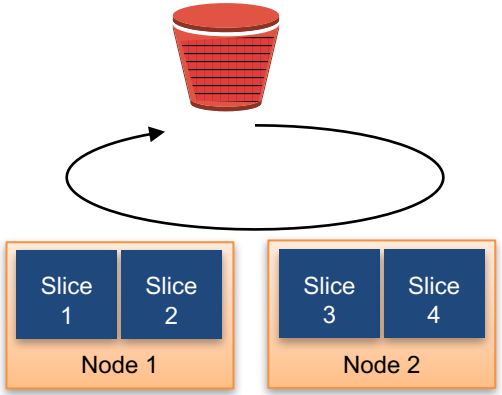
All

All data on every node



Even

Round robin distribution

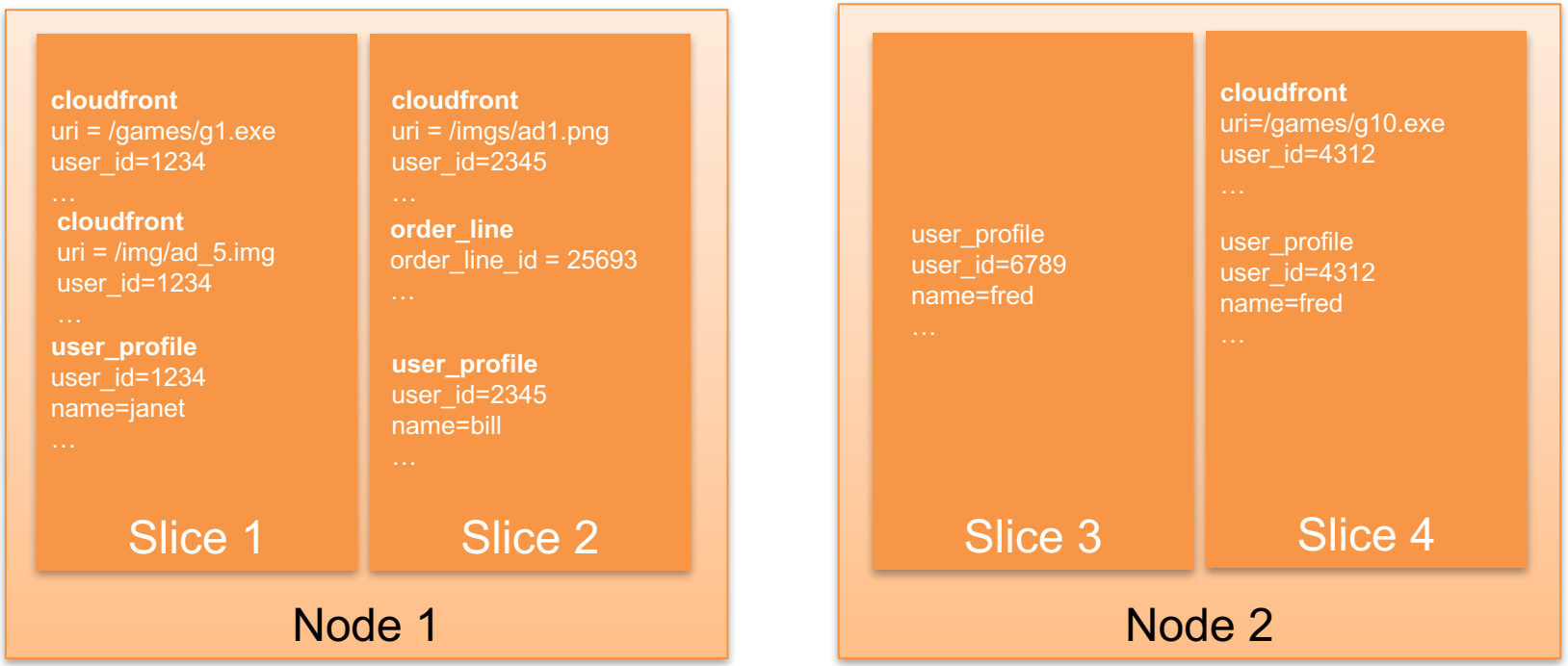


amazon web services

Choosing a Distribution Style

- Choose a distribution style of **KEY** for
  - Large data tables, like a FACT table in a star schema
  - Large or rapidly changing tables used in joins or aggregations
  - Improved performance even if the key is not used in join column
- Choose a distribution style of **ALL** for tables that
  - Have slowly changing data
  - Reasonable size (i.e., few millions but not 100's of millions of rows)
  - No common distribution key for frequent joins
  - Typical use case – joined dimension table without a common distribution key
- Choose a distribution style of **EVEN** for tables that are not joined and have no aggregate queries

Data Distribution with Distribution Keys

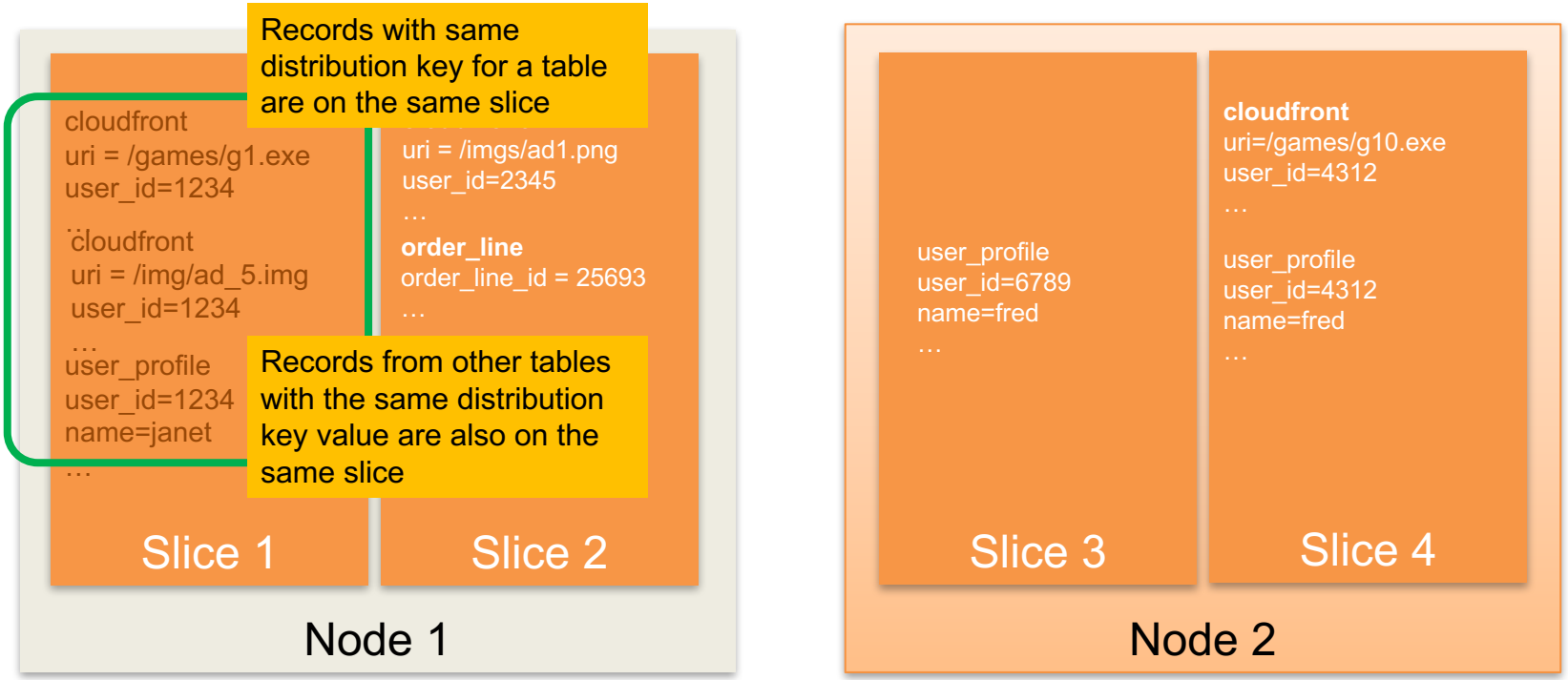


Data Distribution and Distribution Keys

Distribution Keys determine which data resides on which slices

amazon web services

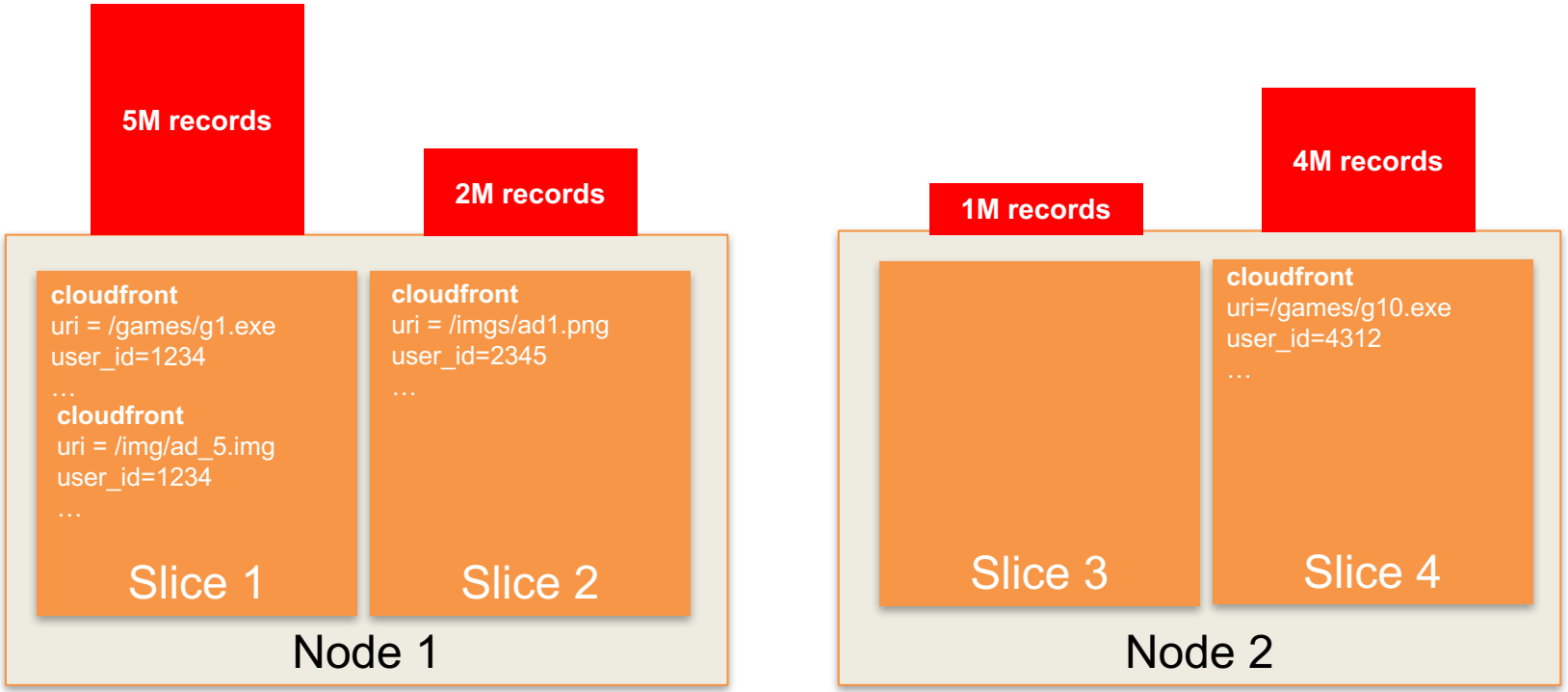
Data Distribution and Distribution Keys



Distribution Keys help with data locality for join evaluation

amazon web services

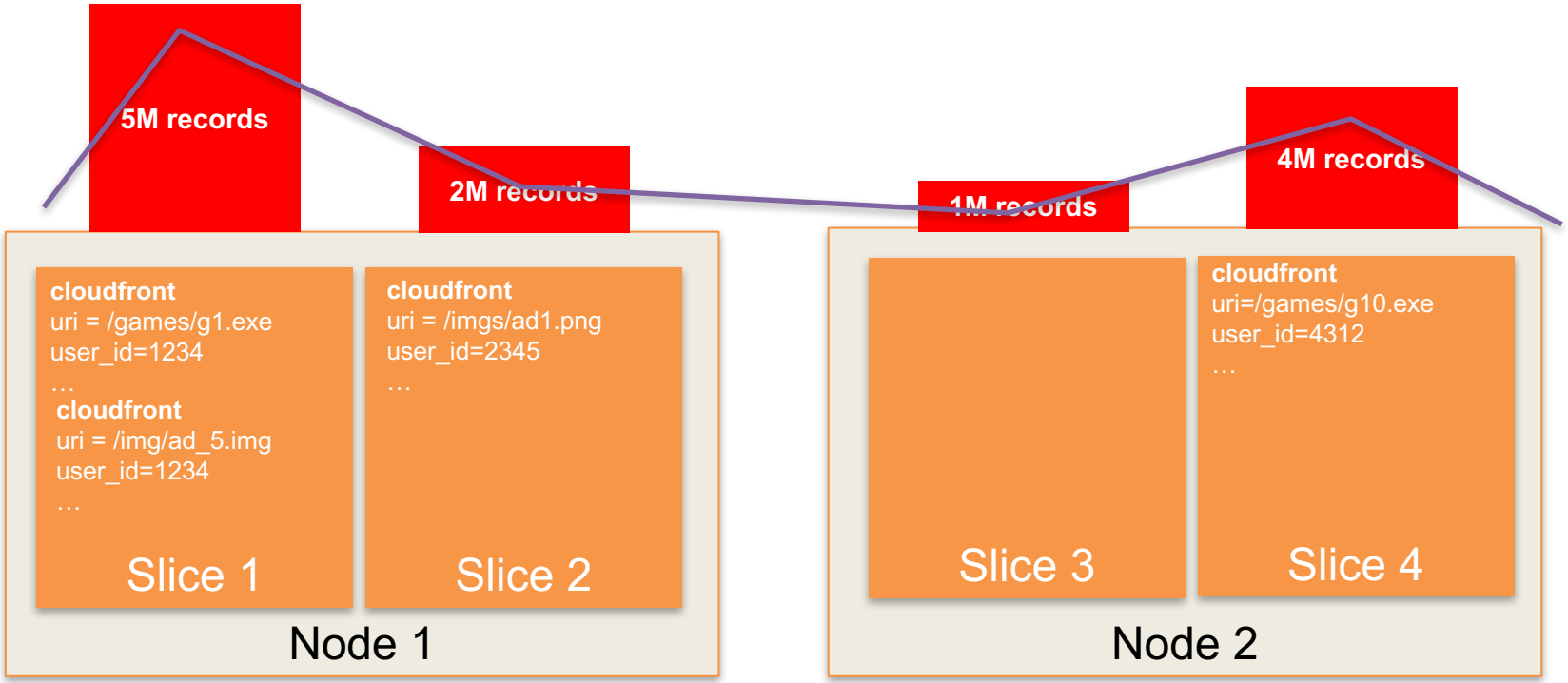
Data Distribution and Distribution Keys



Poor key choices lead to uneven distribution of records…

amazon web services

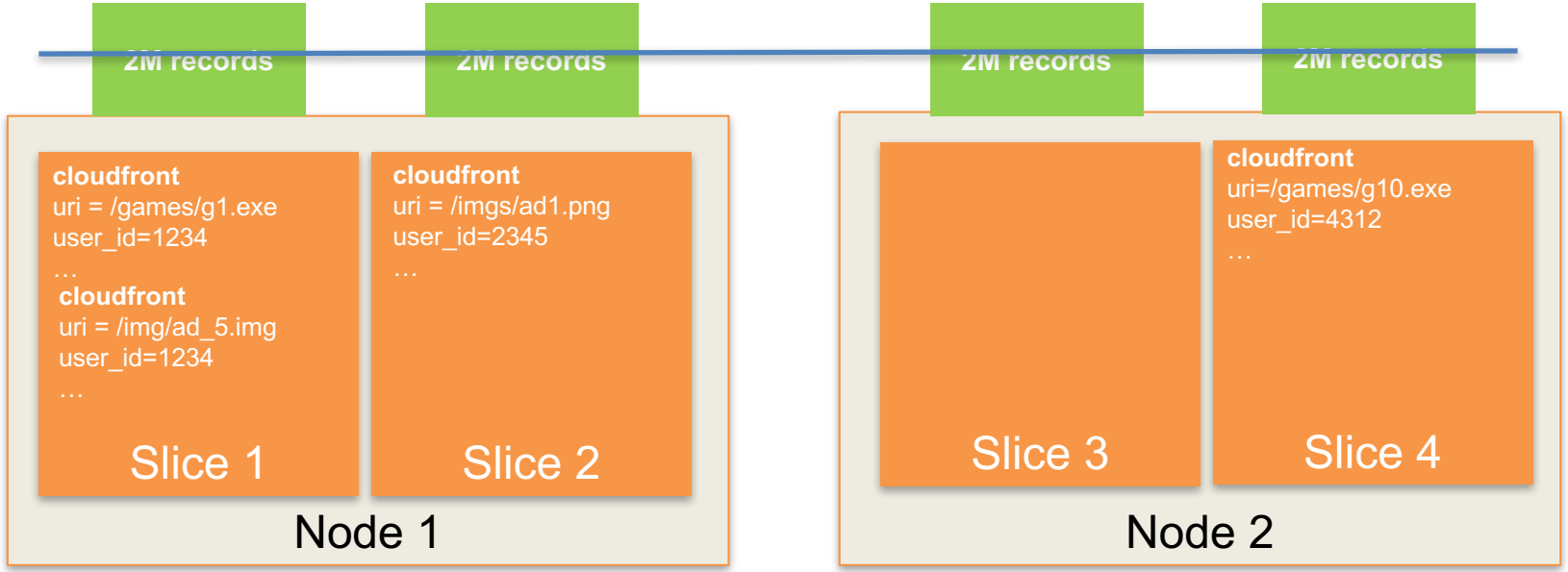
Data Distribution and Distribution Keys



Unevenly distributed data cause processing imbalances!

amazon web services

Data Distribution and Distribution Keys



Evenly distributed data improves query performance

amazon web services

Choosing a Good Distribution Key

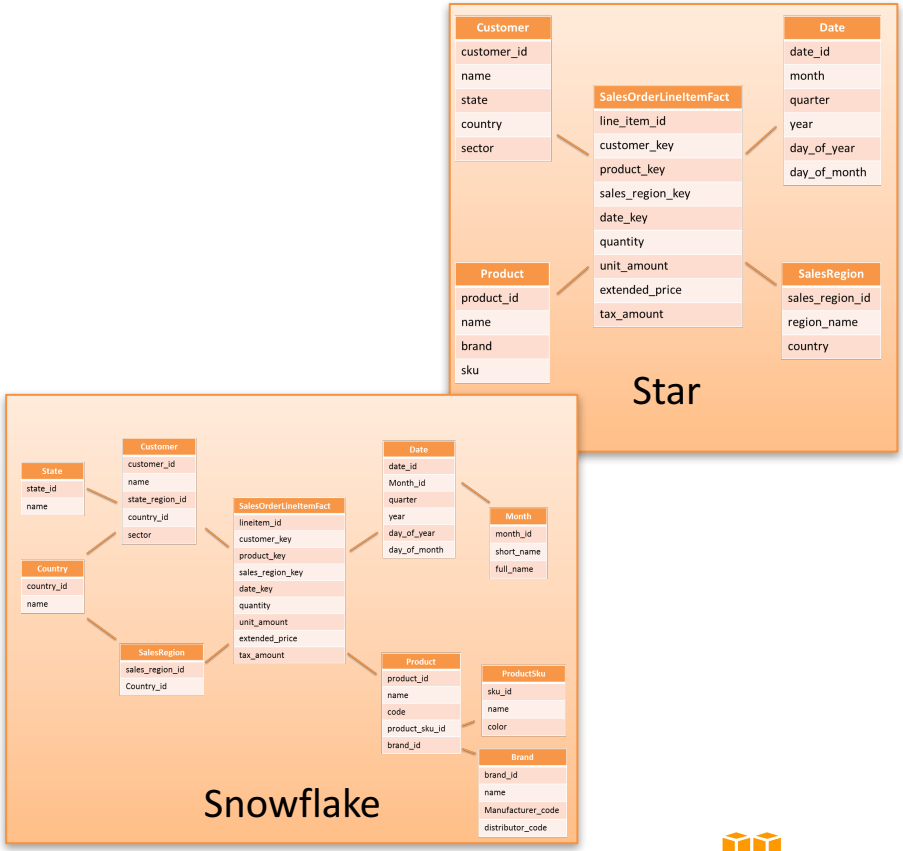
- **Goal**
  - Distribute data evenly across nodes
  - Minimize data movement: Co-located Joins & Aggregates
- **Best Practice**
  - Use the joined columns for largest commonly joined tables as key (example: fact table and large dimension table)
  - Consider using Group By column as a key (GROUP BY clause)
  - Never use a distribution key that causes severe data skew
  - Choose a key with high cardinality; large number of discrete values
- **Avoid**
  - Keys used as equality filter as your distribution key (Concentrates processing on one node)

Star/Snowflake Schemas Overview

- Star and snowflake schemas organize descriptive information around a central fact table that contains measurements

- Fact tables:
  - have multiple foreign keys
  - each foreign key relates to a dimension table

- Snowflake schemas further normalize dimension tables



amazon web services

Data Distribution for Star Schemas

- Choose the foreign key for the largest, most frequently joined dimension and use that as a candidate distribution key
  - Check Skew!
  - To see skew, run this script:https://github.com/awslabs/amazon-redshift-utils/blob/master/src/AdminScripts/table_inspector.sql
- Evaluate remaining dimensions
- For slowly changing dimensions, DISTSTYLE=ALL
- For frequently updated dimensions, consider setting a distribution key to spread the work evenly across the cluster

https://blogs.aws.amazon.com/bigdata/post/Tx1WZP38ERPGK5K/Optimizing-for-Star-Schemas-and-Interleaved-Sorting-on-Amazon-Redshift.

amazon web services

Sorting Data

- The sort key helps Redshift minimize I/O
  - For example, a table sorted on timestamp and queried on date range will skip all blocks not in the query range

- In the slices (on disk), the data is sorted by a sort key
  - If no sort key exists Redshift uses the data insertion order

- Choose a sort key that is frequently used in your queries
  - Primarily as a query predicate (date, identifier, …)
  - Optionally choose a column frequently used for aggregates
  - Optionally choose same as distribution key column for most efficient joins (merge join)

- Don't use too many columns per table as sort keys

Sort in multiple flavors!

- **Compound Sort:**
  - Nested collection of columns
  - Data is sorted by the first, and the next column is used as a 'tie breaker' among columns that share values in the leftmost columns. As a general rule, try to use no more than six columns
  - Useful when a sort column is in a query predicate and to make group by and order by more efficient

- **Interleaved Sort:**
  - A recently introduced feature that allows for up to 8 columns, treated as equally important rather than used as a 'tie breaker' as they are in compound sorts
  - Good for Ad Hoc queries: when you do not know which combination of columns you will use to query your data.
  - Shines with larger workloads (100m rows +)

Example – Compound Sort vs Interleaved Sort

- Product (**type**, **color**, **size**)

- Compound sort on **type**, **color**, **size**
  - Where **type** = 'shirt' and **color**='red' and **size** = 'xl': *full advantage of sort*
  - Where **type** = 'shirt' and **color**='red': *some efficiency*
  - Where **type** = 'shirt' and **size** = 'xl': *limited efficiency depending on cardinality of type column*

- Interleaved sort on **type**, **color**, **size**
  - Using any of **type**, **color** and **size** in a where clause will improve performance

Example – Distribution and Sort Keys

```
SELECT  SUM( S.Price * S.Quantity )
FROM SALES S
JOIN CATEGORY C   ON C.ProductId = S.ProductId
JOIN  FRANCHISE  F ON F.FranchiseId = S.FranchiseId
Where C.CategoryId = 'Produce'  And  F.State = 'WA'
AND S.Date Between '1/1/2015'  AND '1/31/2015'
```

Dist key (S) = ProductID

Dist key (C) = ProductID

Dist key (F) = FranchiseID

Sort key (S) = Date

-- Total Products sold in Washington in January 2015

amazon web services

Choosing a Column Compression Type

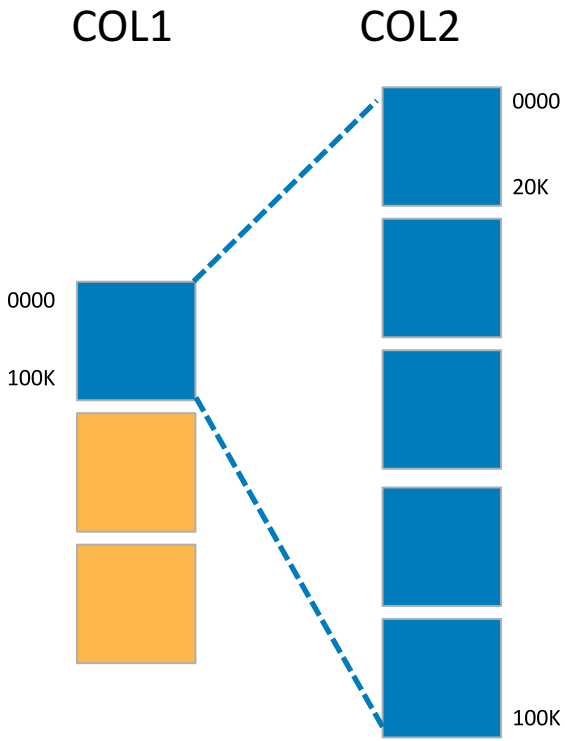
- Columnar compression delivers increased performance and lower cost
- The “COPY” command automatically analyzes and compresses data when loading into empty tables
- The “ANALYZE COMPRESSION” command checks existing tables and proposes optimal compression algorithms for each column
- You can query system tables for storage utilization
- You cannot change the compression encoding for a column after the table is created
- You can specify the encoding for a column when it is added to a table using the ALTER TABLE command

Compressing Data: algorithms

Encoding type	Keyword in CREATE TABLE and ALTER TABLE	Data types	Comment
Raw	RAW	All	No compression
Byte dictionary	BYTEDICT	All except BOOLEAN	Uses a dictionary for repeating values
Delta	DELTA	SMALLINT, INT, BIGINT, DATE, TIMESTAMP, DECIMAL	Optimized for consecutive or closely related values (datetimes, sequence, …)
	DELTA32K	INT, BIGINT, DATE, TIMESTAMP, DECIMAL	
LZO	LZO	All except BOOLEAN, REAL, and DOUBLE PRECISION	Optimal for long character strings, e.g., product descriptions, user comments, or JSON strings
Mostlyn	MOSTLY8	SMALLINT, INT, BIGINT, DECIMAL	Useful for values concentrated in the low end of the spectrum
	MOSTLY16	INT, BIGINT, DECIMAL	
	MOSTLY32	BIGINT, DECIMAL	
Run-length	RUNLENGTH	All	Optimal for large contiguous sets of identical values
Text	TEXT255	VARCHAR only	For text values, uses a dictionary to compress words in the text
	TEXT32K	VARCHAR only	

Be Careful When Compressing Your Sort Keys

- Zone maps store min/max per block
- After we know which block(s) contain the range, we know which row offsets to scan
- Highly compressed sort keys means many rows per block
- You'll scan more data blocks than you need
- If your sort keys compress significantly more than your data columns, you may want to skip compression of sortkey column(s)
- Check SVV_TABLE_INFO(skew_sortkey1)



amazon web services

Defining constraints

- Redshift does not enforce constraints (not null, primary key, foreign key, unique values)
- Redshift uses keys and uniqueness constraints as optimization hints for the query planner
- Define constraints for the optimizer
- Redshift assumes the data provided follows the constraints placed
  - Ingesting data that violates these constraints could lead to erroneous results
- Analyze frequently, minimally on the sort, dist and constraint columns

https://github.com/awslabs/amazon-redshift-utils/tree/master/src/AnalyzeVacuumUtility.

Primary Keys and Manifest Files

- Amazon Redshift doesn't enforce primary key constraints
  - If you load data multiple times, Amazon Redshift won't complain
  - If you declare primary keys in your DML, the optimizer will expect the data to be unique

- Use manifest files to control exactly what is loaded and how to respond if input files are missing
  - Define a JSON manifest on Amazon S3
  - Ensures that the cluster loads exactly what you want

Copying Tables – Best Practices

- Why Copy Tables?
  - Transforming data using SQL, update compression, or change distribution / sort keys
- Create table
  - Intentional about table structure, constraints, compression
  - Generally preferred as greatest level of control
- Create table X (like Y)
  - Creates a empty table matching the original
  - Follow with Insert X as select * from Y
  - Use if you want exactly what you have already got (example, time series tables)
- Create Table As…(CTAS)
  - Distribution and sort keys are inherited where possible if the CTAS statement does not define its own keys
  - Does not inherit compression encodings, constraints, identity columns, default column values, or the primary key from the source table
- Alter Table Append
  - moves only identical columns from the source table to the target table
  - Usually much faster than a similar CREATE TABLE AS or INSERT INTO operation because data is moved, not duplicated
- Prefer use of temp tables for intermediate results

Optimizing a database for querying

- Make sure your columns are compressed appropriately
- Co-locate frequently joined tables using distribution keys or distribution all to avoid data transfers between nodes
- For joined tables consider using sort keys on the joined columns, allowing fast merge joins
- Compression allows you to de-normalize without penalizing storage, simplifying queries and limiting joins
- Vacuum and Analyze regularly