Amazon Redshift Tuning

Jeremy Winters, Pankaj Batra

www.full360.com

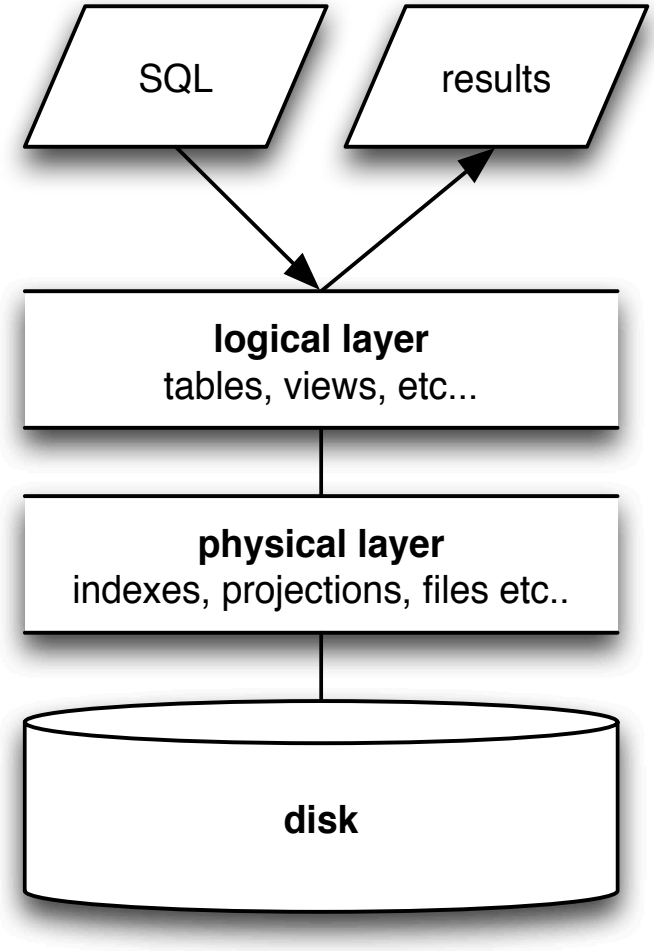
Agenda

- Personal Introductions
- Discuss Class Agenda
- Intro to Full360
- Establish Connectivity
- Columnar vs. Row Based discussion
- Break
- Lab 1 – Merge join optimization
- Compression/encoding discussion
- Lab 2 – Compression analysis
- Lunch
- Discuss Predicate Pruning
- Lab 3 – Optimize for Predicates
- Break
- Interleaved Sortkeys
- Lab 4 – Interleaved Sortkeys
- Q&A
- Thanks for coming!

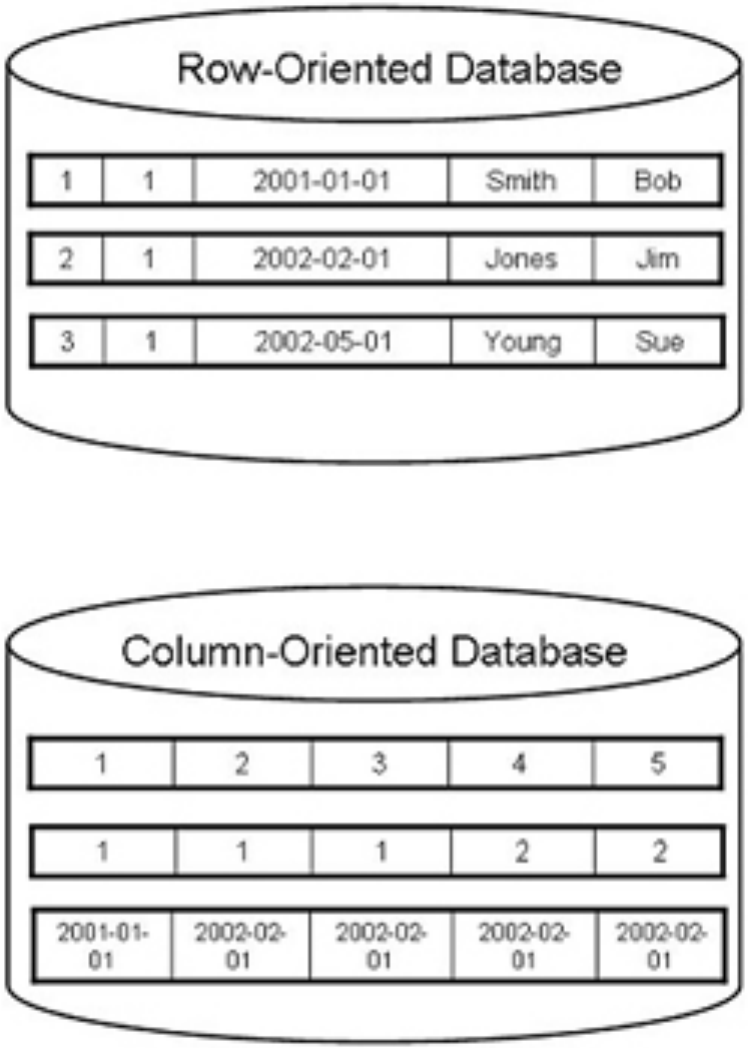
Row vs. column store overview

Some basics…

- Tables represent data sets stored in a row by column format (you knew this…)
- Tables are a *logical* construct based on the SQL "standard"… allowing for a consistent data model definition from a user perspective.
- Physical layer is the specific RDBMS implementation of the logical layer from a software perspective.
- The physical layer allows for a variety of storage patterns. These patterns can be tailored for different use cases such as:
  - transactions (row store)
  - analytics (column store)
  - real-time analytics (in-memory)



Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Sternle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura



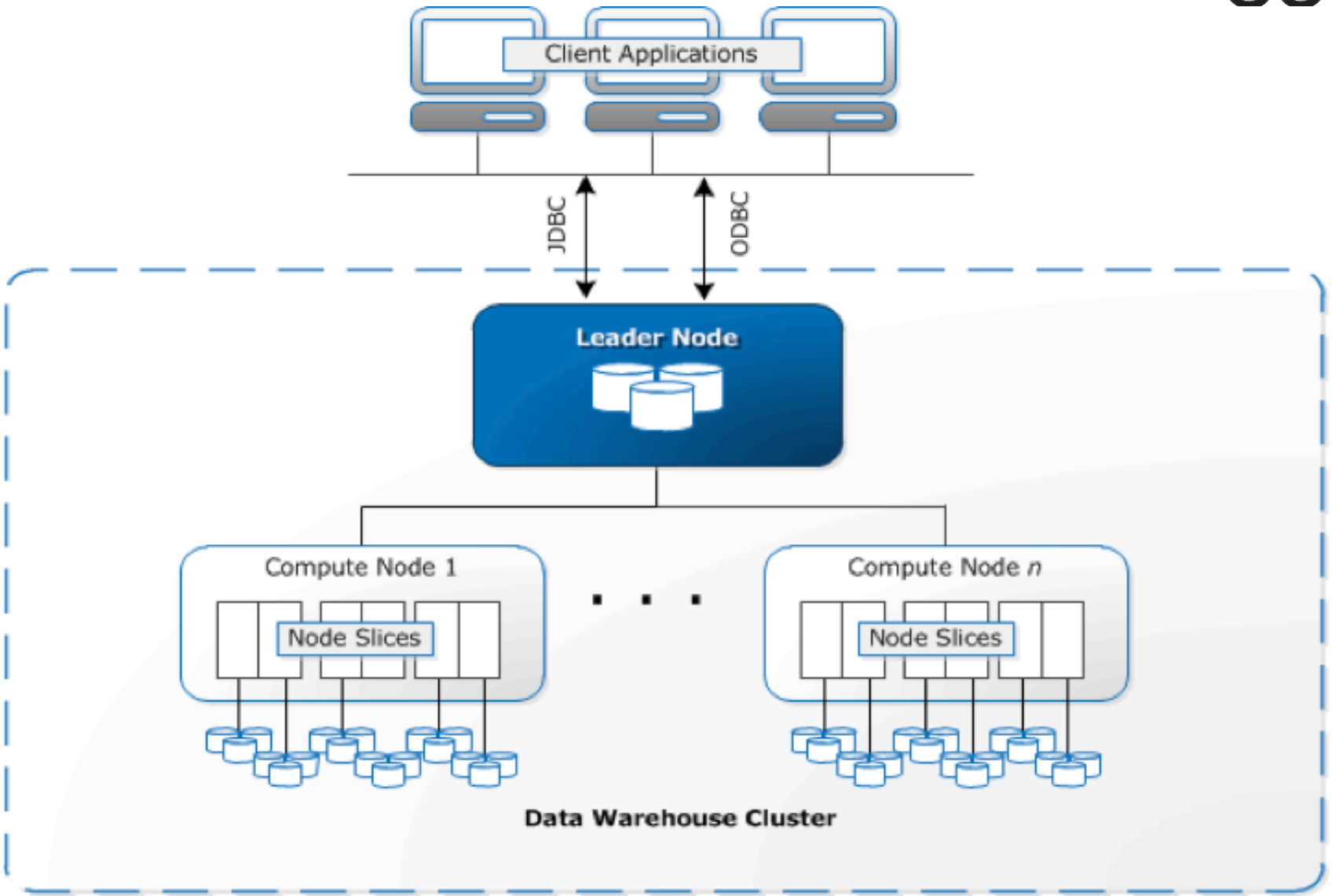
Row based storage

- Row based storage keeps all data for a given row in a contiguous region on the disk.
- This storage pattern is good for OLTP systems which frequently handle transactions involving the select/delete/update of individual records.
- Indexes are used to quickly identify a given record based upon the value of a given field.

Column based storage

- Each field in the table is stored in a separate set of files.
- Storing fields separately allows for easy exclusion of columns which are not required for a given query.
- Sorting the table facilitates the correlation between columns… as well as local joins and aggregations.
- Each column can be encoded or compressed in a way that is appropriate to the data and data type… reducing I/O overhead as the data is read from disk.
- Data can be distributed across all slices allowing for high parallelism at query run time.
- Select * is the most inefficient thing you can do…

Redshift System Overview - Database Architecture



Break

Lab 1 – Merge Joins

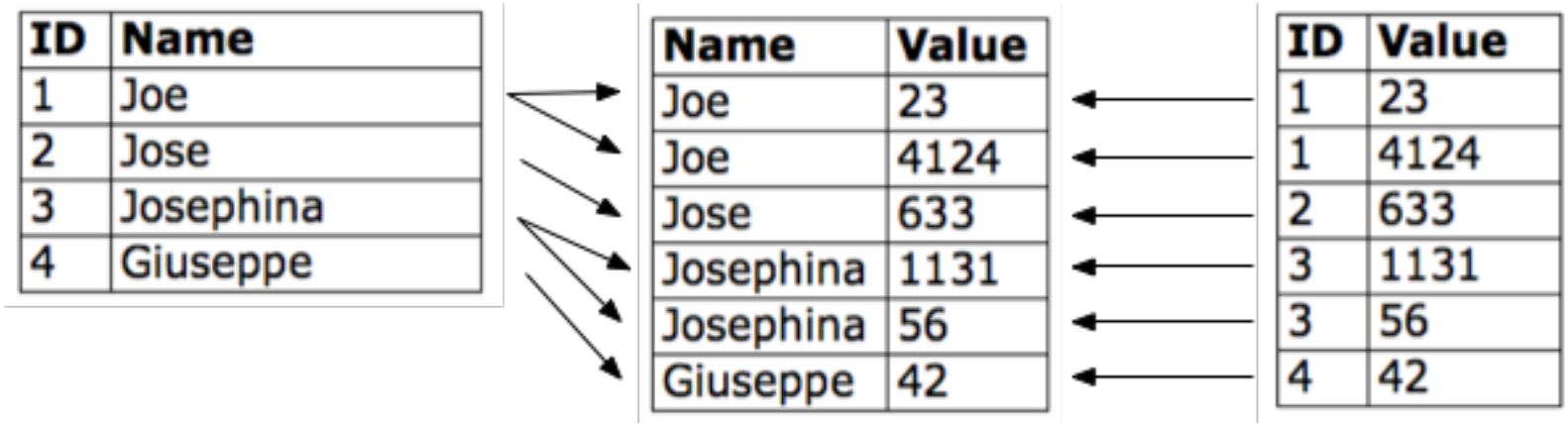
Explain Plans

- Use the keyword EXPLAIN before any query to see the execution plan created by the server.
- Each operation is detailed from bottom to top… right to left.
- Explain plans are daunting at first… so start by looking for expensive operations:
  - DS_DIST_ALL_INNER
  - DS_BCAST_INNER
  - DS_DIST_ALL
  - Nested Loop (cartesian join)
- http://docs.aws.amazon.com/redshift/latest/dg/c_data_redistribution.html

Merge Join

- Merge join is possible when two tables have the following conditions:
  - Identical sortkeys
  - Identical distkeys (or one table DISTSTYLE ALL)
  - Less than 20% unsorted
- Merge joins are efficient because a hash table does not need to be created at run time… the data simply streams through memory and joins like a weird looking zipper.
- https://en.wikipedia.org/wiki/Sort-merge_join

```
select
    a.name
    ,b.value
from
    a join b
        on
            a.id = b.id
```



ID	Name
1	Joe
2	Jose
3	Josephina
4	Giuseppe

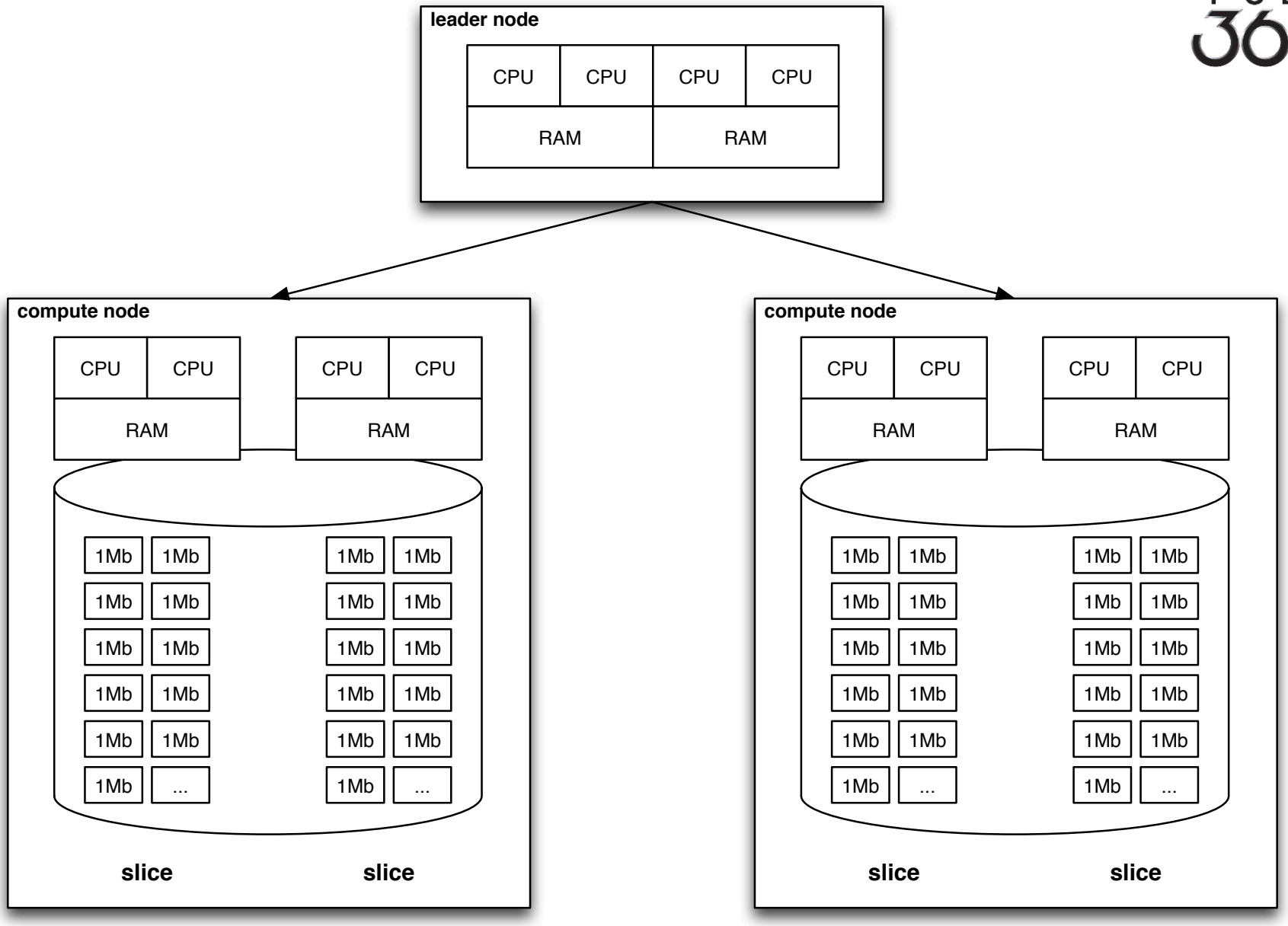
Name	Value
Joe	23
Joe	4124
Jose	633
Josephina	1131
Josephina	56
Giuseppe	42

ID	Value
1	23
1	4124
2	633
3	1131
3	56
4	42

Compression and Encoding

Redshift – Cluster to Block

- Redshift has the following hierarchy
  - **Cluster** – A group of servers comprising a leader node and at least 2 compute nodes
  - **Node** – An AWS machine instance
  - **Slice** – A set of CPUs and RAM. 2-8 slices on a node… used to manage a set of blocks
  - **Block** – 1MB disk blocks managed within a given slice… file system is optimized for 1MB disk blocks


leader node
CPU
CPU
CPU
CPU
RAM
RAM
compute node
CPU
CPU
CPU
CPU
RAM
RAM
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
...
1Mb
...
slice
slice
compute node
CPU
CPU
CPU
CPU
RAM
RAM
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
1Mb
...
1Mb
...
slice
slice

Encoding

- By default, all data stored in the 1MB disk blocks is RAW… no encoding or compression applied
- Encoding compresses the size of a given column so that it requires less blocks to store the data on the disk
- Even though Redshift is optimized for 1MB blocks… disk access is still the slowest operation
- Having less blocks on disk means that less disk access operations are required for pulling the data into memory
- Encoding types must be specified in the CREATE TABLE statement, and can not be changed after the table is created

Encoding Types

- **RAW** – default… uncompressed… no encoding
- **BYTEDICT** – each block contains a small dictionary with all unique values in the block. Individual values are stored as an index to the dictionary
- **DELTA/DELTA32K** – for integer based data types… data is stored as the delta from one row to the next. Good for sorted columns.

Encoding Types

- **LZO** – fast compression algorithm… good for many things including large bodies of text
- **MOSTLY8/16/32** – For 16-64 bit integers… stores the value as an 8/16/32 bit by default. Useful for when most data types are significantly smaller than the max allowed by data type
- **RUNLENGTH** – excellent for when values repeat from row to row in medium to large "runs"

Encoding Types

- **TEXT255/TEXT32K**– Similar to BYTEDICT but focused on repeated words within text fields.

- Note that not every encoding will work with every data type.

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

Encoding Types

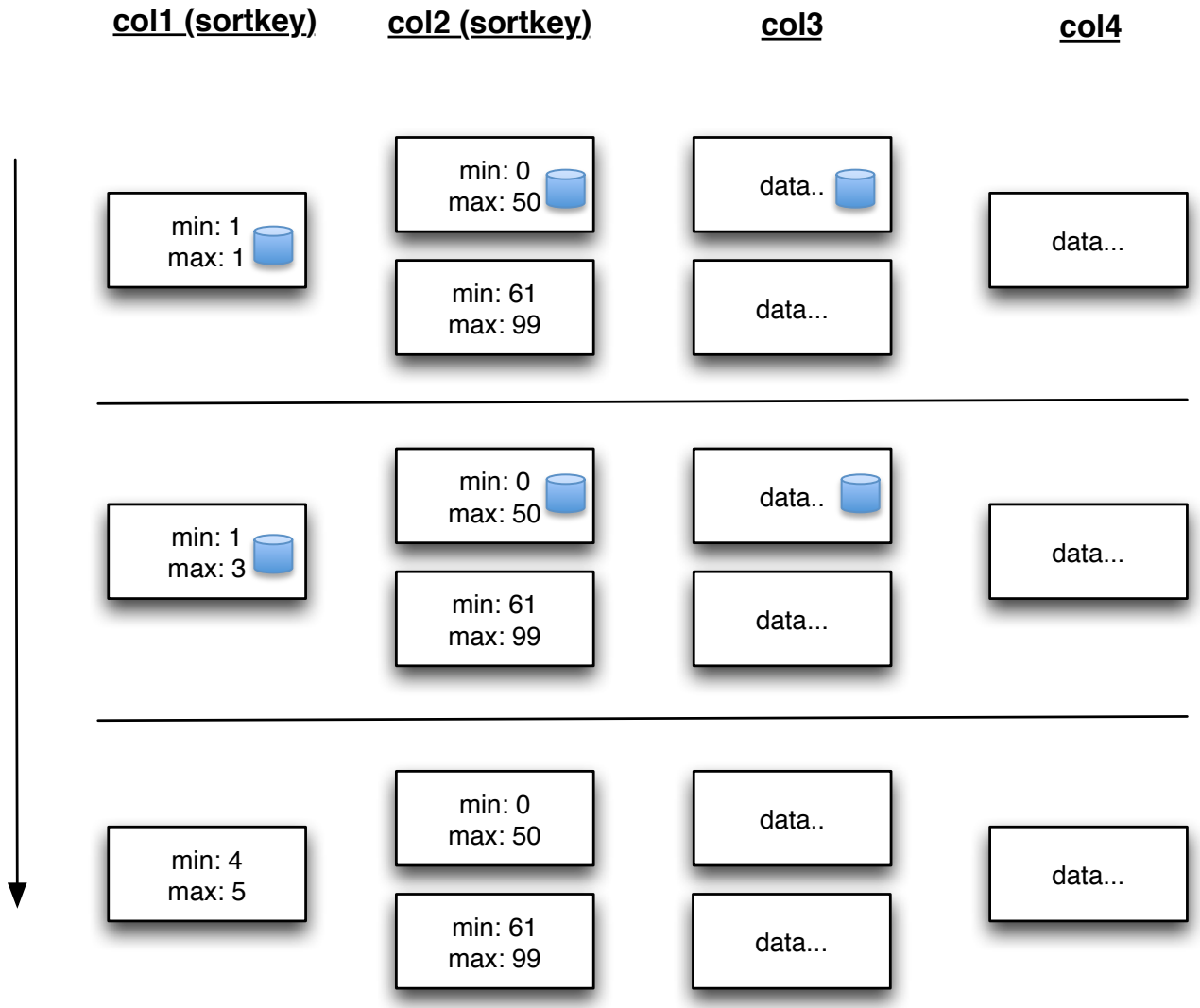
Encoding type	Keyword in CREATE TABLE and ALTER TABLE	Data types
Raw (no compression)	RAW	All
Byte dictionary	BYTEDICT	All except BOOLEAN
Delta	DELTA DELTA32K	SMALLINT, INT, BIGINT, DATE, TIMESTAMP, DECIMAL INT, BIGINT, DATE, TIMESTAMP, DECIMAL
LZO	LZO	All except BOOLEAN, REAL, and DOUBLE PRECISION
Mostly*n*	MOSTLY8 MOSTLY16 MOSTLY32	SMALLINT, INT, BIGINT, DECIMAL INT, BIGINT, DECIMAL BIGINT, DECIMAL
Run-length	RUNLENGTH	All
Text	TEXT255 TEXT32K	VARCHAR only VARCHAR only

Lab 2 – Compression Effectiveness

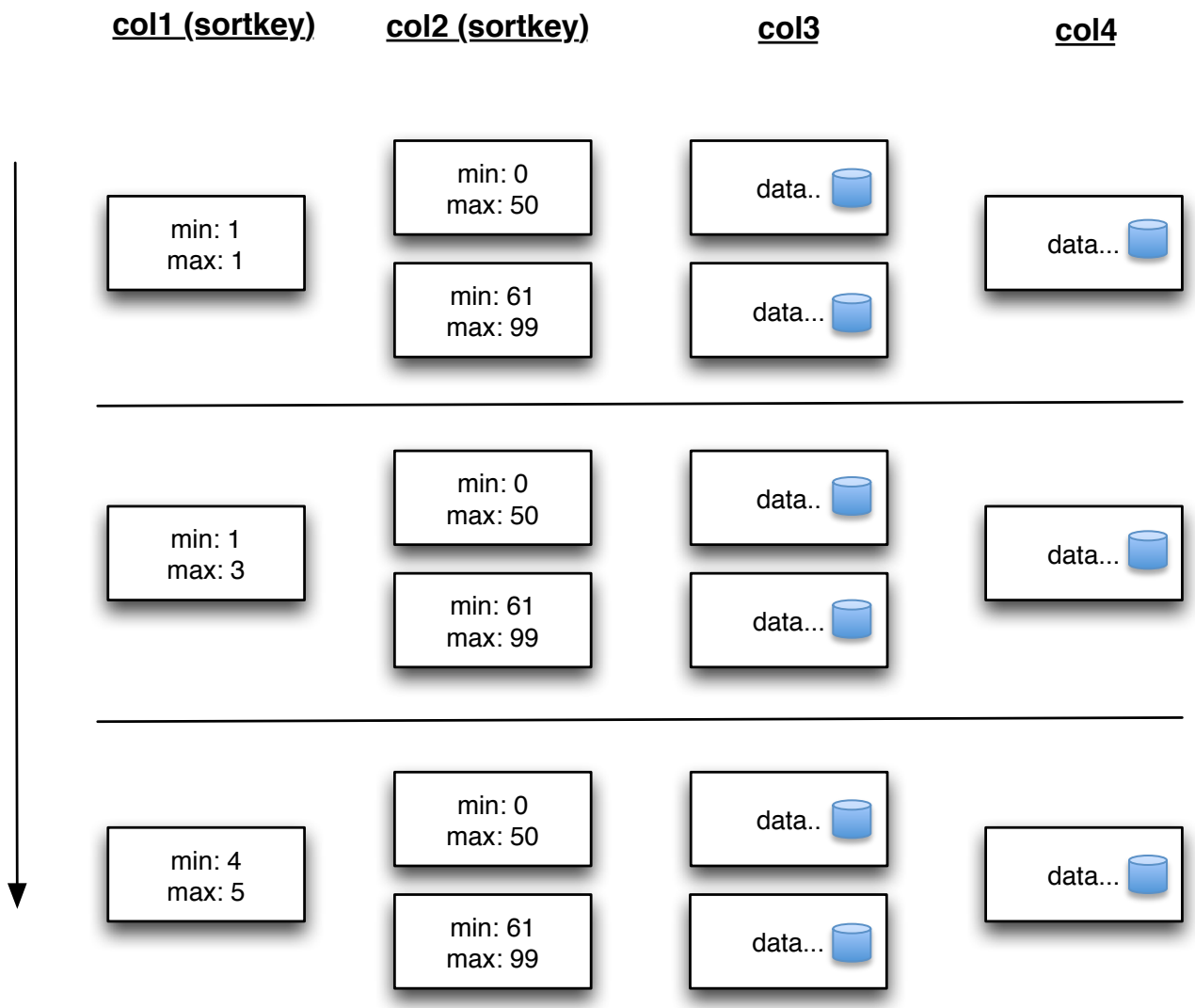
Predicate Pruning with Sortkeys

Sortkeys

- Columns specified as SORTKEY will be used to sort the entire table
- Redshift uses the hidden ROW_ID column to store the correlation between the blocks so that it can determine which rows are contained within each block
- Each sortkey block stores minimum and maximum values contained within… which can be used at query time to identify whether or not a block potentially contains the values in the where clause
- At query time… Redshift can use this approach to "prune" the number of blocks pulled from disk based upon the query predicates



select col3 from table where col1 = 1 and col2 = 40;



select col4 from table where col3 = ‘turkey’;

Lab 3 – Predicate Pruning

Interleaved Sortkeys

Sortkey comparison

- Compound – Default for Redshift… The first column specified in the sortkey takes precedence over the others. If you use the top level keys in most queries, you will get good performance most of the time… but when you run queries that do not filter on the sortkeys you end up with full table scans that can take a long time.
- Interleaved – Instead of a hierarchy of sortkeys… you can specify up to 8 columns which have "equal" precedence in the sorting. What this really ends up providing is a multidimensional index.

Use Cases

- Use cases for interleaved sortkeys
  - OLAP style reporting where the user does not always filter using the same column
  - Very large tables which require many lookups of small bits of information
  - Slowly changing data sets… archives… weekly builds

Lab 4 – Interleaved Sortkeys