Advanced Database Systems (INFS3200)

Lecture 8: Column Oriented Databases

Lecturer: Dr Zhi Chen

School of Mathematics, Physics and Computing

The University of Southern Queensland

Dr Zhi Chen

- Lecturer in Computing (**INFS3208, DATA7002, INFS3200**)
- Research Interests: Computer Vision, Generative Models
- Homepage: http://uqzhichen.github.io/
- Email: zhi.chen@unisq.edu.au
- Zoom: https://uqz.zoom.us/j/2352620476
- Office: Room D102, Block D, Toowoomba campus

Outline

- Background and Concepts about Column-Oriented Databases
- ClickHouse - An open-source column-oriented DBMS
- Column-Oriented Database Compression
  - Dictionary Encoding
  - Run-length Encoding
  - Delta Encoding
  - Bit-Packing Encoding
- Optimizing Query Performance
  - Sparse Primary Index
  - Data Skipping Index
  - Materialized View
  - Projection
  - Parallelism

Background – Online Analytical Processing (OLAP)

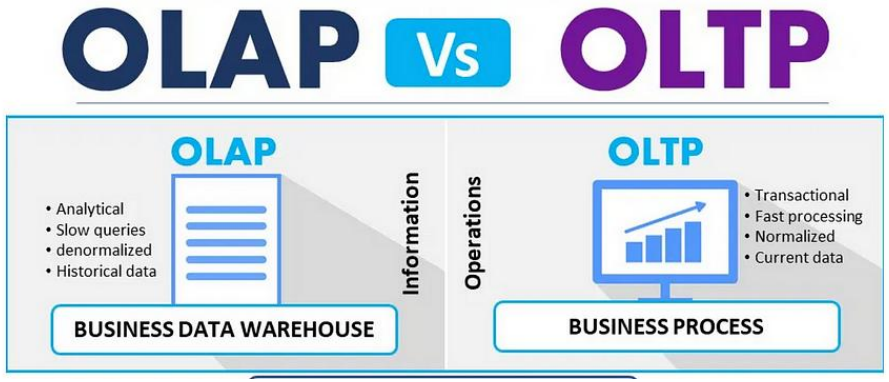
- Online Analytical Processing is an approach to ***quickly answer multi-dimensional analytical queries***.
- The term OLAP was created as a slight modification of the traditional database term online transaction processing (OLTP)
- Milestones:
  - Express Database (1970s), Now owned by Oracle
  - Essbase introduced by Arbor Software Company in 1992.
  - The terms **OLAP was first defined** by *Edgar F. Codd* (“the father of the relational database”) in 1993.
  - In 1998, Microsoft released its first OLAP Server – Microsoft Analysis Services, which drove wide adoption of OLAP technology and moved it into the mainstream.
  - 2006 - Palo launched - The first open-source OLAP server.

Palo by Jedox

https://en.wikipedia.org/wiki/Online_analytical_processing#cite_note-28
Codd E.F.; Codd S.B. & Salley C.T. (1993). "Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate“
https://web.archive.org/web/20071221044811/http://www.olapreport.com/origins.htm

Concepts - Database Management Systems (DBMS)

- ***OLTP – Online Transactional Processing***
  - Fast operations reading small amount of data each time
  - Continuous stream of transactions modifying current state of data
- ***OLAP – Online Analytical Processing***
  - Building reports based on large volume of historical data
  - Compute aggregates through complex queries
- Hybrid Transactional/Analytical Processing



https://medium.com/@sounder.rahul/oltp-olap-systems-for-beginners-data-engineers-b3d6af741472

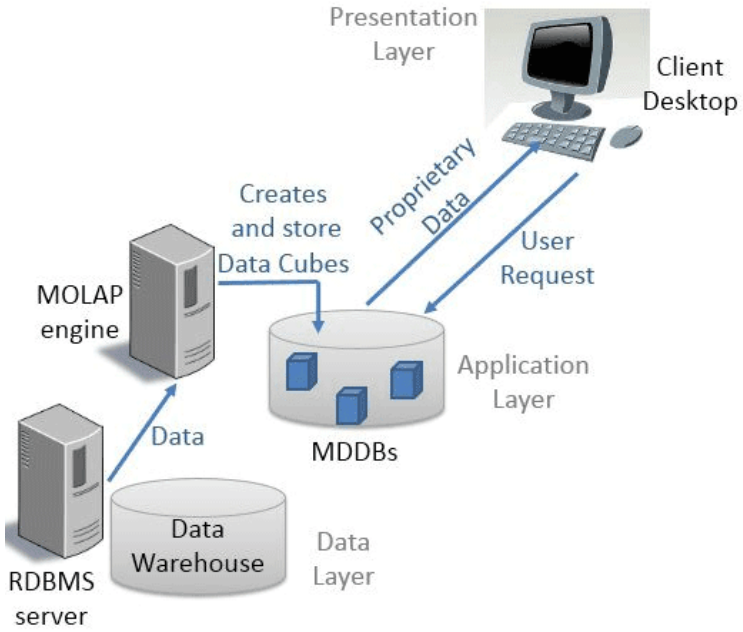
Concepts – OLTP vs OLAP

Criteria	OLAP	OLTP
Purpose	**Analyse large volumes of data** to support decision-making.	**Manage and process real-time transactions.**
Data source	Use historical and aggregated data from multiple sources.	Use real-time and transactional data from a single source.
Data structure	**Multidimensional** or **relational** databases.	**Relational** databases.
Data model	Star schema, snowflake schema, or other analytical models.	Normalized or denormalized models.
Volume of data	**Large storage**. (terabytes TBs and petabytes PBs).	**Comparatively smaller storage** (gigabytes GBs).
Response time	**Longer response** times, typically in seconds or minutes.	**Shorter response** times, typically in milliseconds
Example applications	Analysing trends, predicting customer behaviour, and identifying profitability.	Processing payments, customer data management, and order processing.

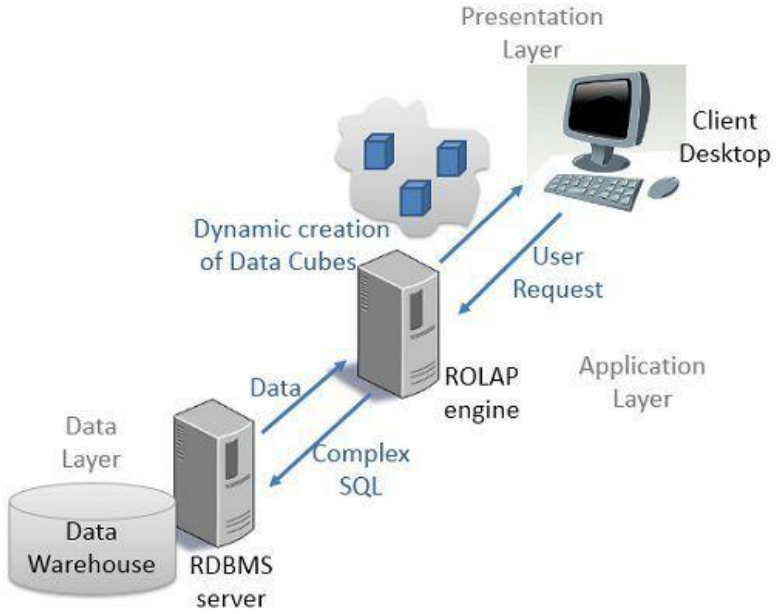
https://aws.amazon.com/compare/the-difference-between-olap-and-oltp/

Concepts – Types of OLAP

- ***Multidimensional OLAP*** (MOLAP) is the classic form of OLAP
  - MOLAP stores the data in an optimized **multi-dimensional array storage.**
  - Products: Microsoft Analysis Services, Essbase, etc.
- ***Relational OLAP*** (ROLAP) works directly with **relational databases** and does not require pre-computation.
  - Products: Oracle OLAP, IBM Cognos Analytics, etc.



MOLAP Model



ROLAP Model

https://www.cloudzilla.ai/dev-education/molap-vs-rolap-vs-holap/

Concepts – Data Orientation

- **Data orientation** refers to how tabular data is represented in a linear memory model such as in-disk or in-memory.

column 1	column 2	column 3
item 11	item 12	item 13
item 21	item 22	item 23

Concepts – Data Orientation

- **Data orientation** refers to how tabular data is represented in a linear memory model such as in-disk or in-memory.

column 1	column 2	column 3
item 11	item 12	item 13
item 21	item 22	item 23

item 11	item 12	item 13	item 21	item 22	item 23

row-oriented

Concepts – Data Orientation

- **Data orientation** refers to how tabular data is represented in a linear memory model such as in-disk or in-memory.

column 1	column 2	column 3
item 11	item 12	item 13
item 21	item 22	item 23

column-oriented

Concepts – Data Orientation

- **Data orientation** refers to how tabular data is represented in a linear memory model such as in-disk or in-memory.
- The two most common representations are ***column-oriented*** (columnar format) and ***row-oriented*** (row format)
  - The choice of data orientation is a trade-off and an architectural decision in databases, query engines, and numerical simulations

column 1	column 2	column 3
item 11	item 12	item 13
item 21	item 22	item 23

item 11	item 21	item 12	item 22	item 13	item 23

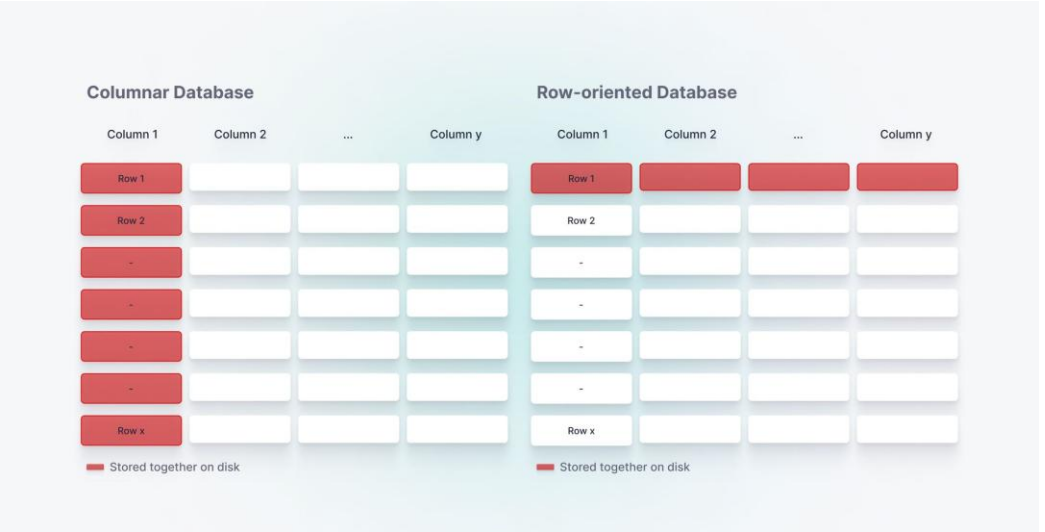
column-oriented

item 11	item 12	item 13	item 21	item 22	item 23

row-oriented

Concepts - Column-Oriented Databases

- Compare ***column-oriented*** databases to ***row-oriented*** databases, in which the contents of a single row are stored together on disk. Row-oriented databases are optimized for transactional, single-entity lookup instead of ***analytics*** over many entities.

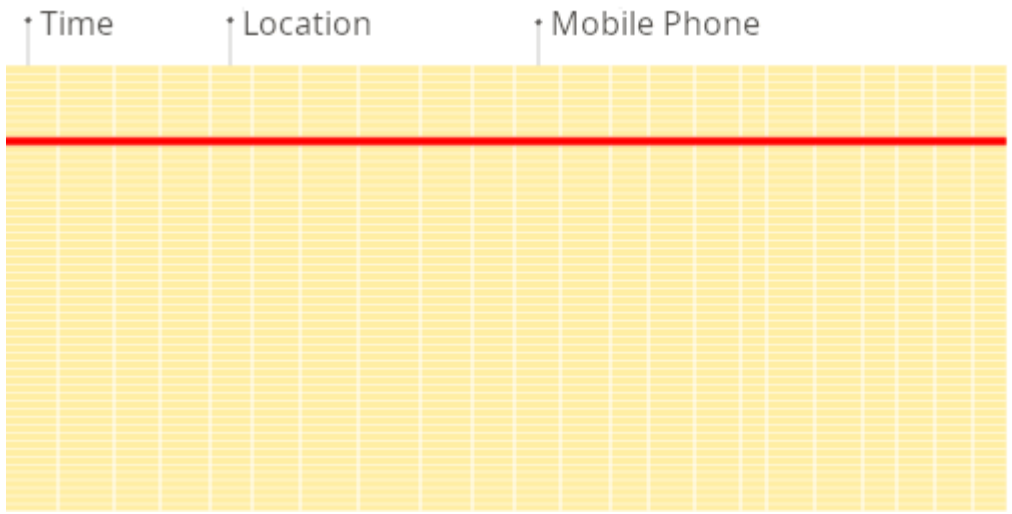


Concepts - Column-Oriented Databases

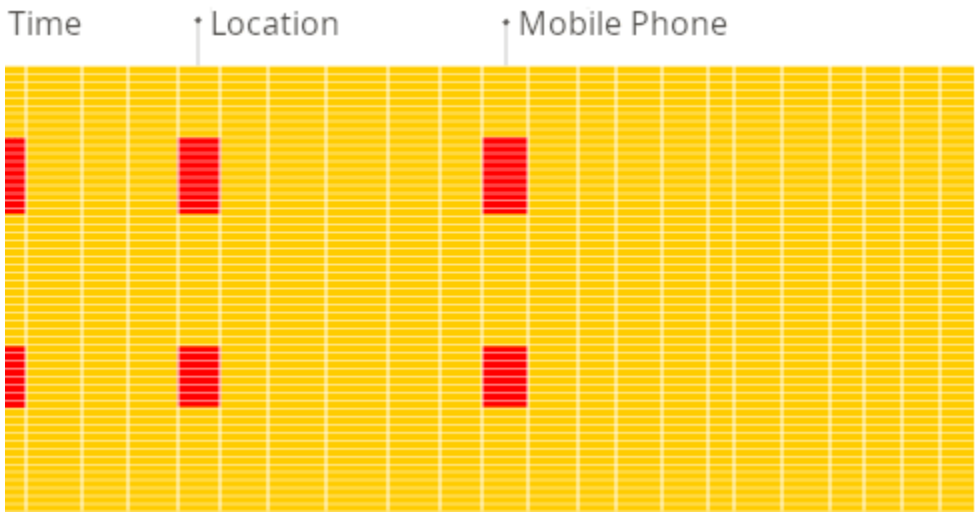
- Definition: A column-oriented database, also known as columnar database, is a type of database management system (DBMS) that ***stores data in columns together*** on disk. This enables faster queries for data analytics, which generally involves ***filtering*** and ***aggregating*** table columns.

Concepts - Column-Oriented Databases

- An straightforward illustration of the advantage of column-oriented databases



Row-oriented



Column-oriented

https://clickhouse.com/docs/en/faq/general/columnar-database

Why Column-Oriented Databases

- A naïve case

user_id	user_name	current_balance	number_of_transactions
1	`simon@gmail.com`	13	81
2	`jason@gmail.com`	256	1
3	`lin@gmail.com`	512	759
4	`yvone@gmail.com`	1024	32
5	`zion@gmail.com`	-32	15

```
SELECT sum(current_balance)
AS total_balance
FROM table
WHERE user_id > 2
```

Why Column-Oriented Databases

- A naïve case – column-oriented database desired

user_id	user_name	current_balance	number_of_transactions
1	`simon@gmail.com`	13	81
2	`jason@gmail.com`	256	1
3	`lin@gmail.com`	512	759
4	`yvone@gmail.com`	1024	32
5	`zion@gmail.com`	-32	15

Only one column access required !

```
SELECT sum(current_balance)
AS total_transactions
FROM table
WHERE user_id > 2
```

Why Column-Oriented Databases

- A naïve case

user_id	user_name	current_balance	number_of_transactions
1	`simon@gmail.com`	13	81
2	`jason@gmail.com`	256	1
3	`lin@gmail.com`	512	759
4	`yvone@gmail.com`	1024	32
5	`zion@gmail.com`	-32	15

```
SELECT user_id, user_name, current_balance
FROM table
WHERE user_id = 2
```

Why Column-Oriented Databases

- A naïve case – row-oriented data storage desired

a single row of data in multiple columns

user_id	user_name	current_balance	number_of_transactions
1	`simon@gmail.com`	13	81
2	`jason@gmail.com`	256	1
3	`lin@gmail.com`	512	759
4	`yvone@gmail.com`	1024	32
5	`zion@gmail.com`	-32	15

```
SELECT user_id, user_name, current_balance,
Number_of_transactionos
FROM table
WHERE user_id = 2
```

Column-Oriented Database Advantages (1/2)

1. **Improved data compression**
   - Data within a single column is typically **homogenous (same data type)**.
   - Amenable to applying advanced compression techniques, reduce storage requirements and associated costs.
   - This compression also results in less I/O overhead, as less data needs to be read from storage.
2. **Enhanced query performance**
   - In a columnar database, **only the columns relevant to a query need to be accessed and processed**.
   - Selective data retrieval leads to faster query performance.
   - Analytical queries typically aggregate or scan large volumes of data.
3. **Efficient use of cache memory**
   - Columnar storage aligns well with **modern processor cache design**. Since columns are stored contiguously, a single cache load can retrieve a large block of relevant data, reducing cache misses and enhancing CPU efficiency.
4. **Vectorization and parallel processing**
   - Columnar databases are well-suited for **vectorized operations**, where the same operation is applied to multiple data points simultaneously.
   - This aligns well with modern central processing units architectures and allows for efficient parallel processing, further speeding up query execution.

Column-Oriented Database Advantages (2/2)

5. Improved analytics and reporting

- Inherently efficient at **aggregating and summarizing** data, operations that are fundamental to analytics and reporting.
- They allow for faster calculations and aggregations across vast datasets, ideal for business intelligence and analytical applications.

6. Better handling of sparse data

- Handle **missing or null values** efficiently by simply not storing any data for those missing values

7. Flexible indexing options

- Columnar databases offer flexible **indexing strategies**. Because each column is stored independently, different indexing techniques can be applied to different columns, depending on the nature of the data and the typical queries performed.

8. Ease of scalability

- Generally easier to **scale horizontally** by adding more servers to handle increased load.
- Particularly beneficial in cloud computing environments where resources can be dynamically adjusted based on demand.

9. Real-time data analytics and updates

- Some modern columnar databases have evolved to allow for real-time analytics and high-speed data ingestion.
- Traditional columnar databases were seen as less suitable for transactional systems.

Column-Oriented Database Limitations (1/2)

1. **Limited suitability for transactional workloads**:
   - Columnar databases are optimized for ***read-heavy analytical queries*** and not for transactional workloads,
   - The columnar storage model can make transactional operations more complex and time-consuming
2. **Higher overhead for writing data**
   - Writing data to a columnar database is generally slower compared to row-based databases.
   - Each data insertion or update may ***require accessing and modifying several distinct column files***
3. **Complexity in handling joins**
   - Reconstructing complete rows to perform joins can be computationally intensive, potentially impacting query performance.
4. **Resource intensive for small queries**
   - The overhead of accessing separate column files may outweigh the benefits, making such databases less suitable for certain types of small-scale operations.

Column-Oriented Database Limitations (2/2)

5. Increased complexity for certain operations

- Operations that span multiple columns or transactions that require **row-level locking**, can be more resource-intensive.

6. Potential storage overhead for variable-length data

- Face **storage inefficiencies** when dealing with **variable-length** data, such as strings or blobs.
- This can lead to increased storage requirements and reduced compression benefits for certain data types.

7. Scalability and concurrency challenges

- Face challenges in environments with high levels of **write concurrency** or transactional processing.

8. Cost considerations

- Given their specialized nature and the resources required for optimal performance, columnar databases can sometimes entail higher costs in terms of software, hardware, and maintenance, especially for large-scale deployments.

Outline

- Background and Concepts about Column-Oriented Databases
- ClickHouse - An open-source column-oriented DBMS
- Column-Oriented Database Compression
  - Dictionary Encoding
  - Run-length Encoding
  - Delta Encoding
  - Bit-Packing Encoding
- Optimizing Query Performance
  - Sparse Primary Index
  - Data Skipping Index
  - Materialized View
  - Projection
  - Parallelism

ClickHouse

- An ***open-source*** column-oriented DBMS for online analytical processing (***OLAP***) that allows users to generate ***analytical*** reports using SQL queries in real-time.
- True column-oriented DBMS. Nothing is stored with the values. For example, constant-length values are supported to avoid storing their length "number" next to the values.
- ***Linear scalability.*** It's possible to extend a cluster by adding servers.
- ***Fault tolerance***. The system is a cluster of shards, where each shard is a group of replicas. ClickHouse uses asynchronous multi-master replication. Data is written to any available replica, then distributed to all the remaining replicas.
- Capability to store and process ***petabytes of data***.
- ***SQL support***. ClickHouse supports an extended SQL-like language that includes arrays and nested data structures, approximate and URI functions, and the availability to connect an external key-value store.
- ***High performance***.
  - Vector calculations are used. Data is not only stored by columns, but is processed by vectors (parts of columns). This approach allows it to achieve high CPU performance.
  - Sampling and approximate calculations are supported.
  - Parallel and distributed query processing is available (including JOINs).

https://en.wikipedia.org/wiki/ClickHouse

Case Study

- UK Property Price Dataset
  - The data is available since 1995, and the size of the dataset in uncompressed form is about 4 GB

```
CREATE TABLE uk_price_default(
price UInt32,
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-detached' = 2, 'detached' = 3, 'flat' = 4, 'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1, 'leasehold' = 2, 'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (postcode1, postcode2, addr1, addr2);
```

Table schema

price	date	postcode1	postcode2	type	is_new	duration	addr1	addr2	street	locality	town	district	county
70000	1995-08-04			terraced	0	freehold	1		PARKHAM ROAD	BRIXHAM	BRIXHAM	TORBAY	TORBAY
43000	1995-04-21			terraced	0	freehold	1		ZAGGY LANE	CALLINGTON	CALLINGTON	CARADON	CORNWALL
60000	1995-01-27			semi-detached	0	freehold	1		PONDSIDE COTTAGES	GRAVELEY	HITCHIN	NORTH HERTFORDSHIRE	HERTFORDSHIRE
54000	1995-08-04			detached	0	freehold	1		DUKE STREET	STANTON	BURY ST. EDMUNDS	ST EDMUNDSBURY	SUFFOLK
31000	1995-02-06			terraced	0	freehold	1		BEACH ROAD	PERRANPORTH	PERRANPORTH	CARRICK	CORNWALL
70000	1995-05-01			semi-detached	0	freehold	1		THE MYRTLES	DARTFORD	DARTFORD	DARTFORD	KENT
12500	1995-06-23			terraced	0	freehold	1		RHOSGOCH	CWM PENMACHNO	BETWS-Y-COED	CONWY	CONWY
15000	1995-08-16			semi-detached	0	freehold	1		KINGSLAND ROAD	BIRKENHEAD	BIRKENHEAD	WIRRAL	MERSEYSIDE
11000	1995-09-29			terraced	0	freehold	1		BLAKEY STREET	BURNLEY	BURNLEY	BURNLEY	LANCASHIRE
9000	1995-05-15			terraced	0	freehold	1		HANGMAN PATH	COMBE MARTIN	ILFRACOMBE	NORTH DEVON	DEVON

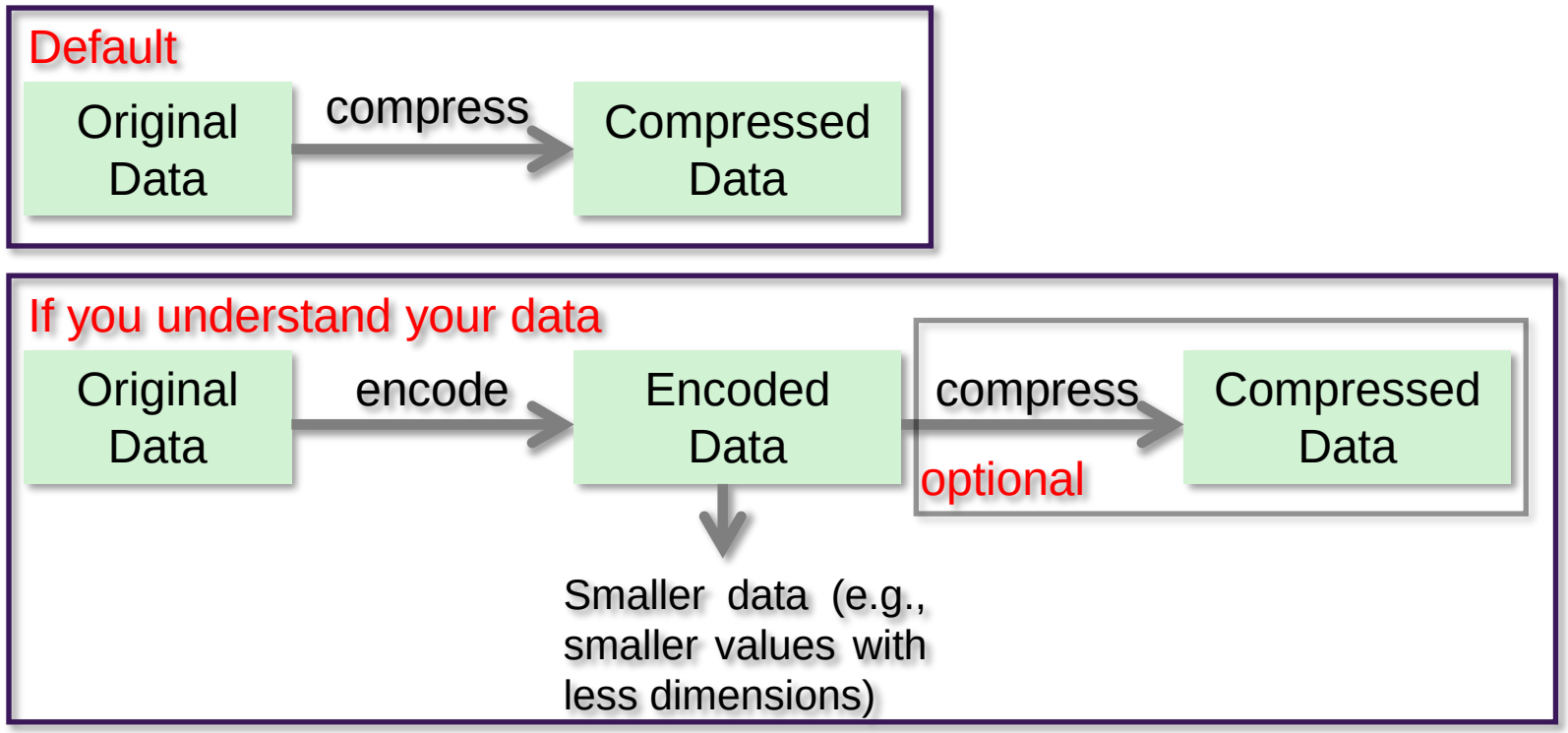
Dataset samples

Outline

- Background and Concepts about Column-Oriented Databases
- ClickHouse - An open-source column-oriented DBMS
- Column-Oriented Database Compression
  - Dictionary Encoding
  - Run-length Encoding
  - Delta Encoding
  - Bit-Packing Encoding
- Optimizing Query Performance
  - Sparse Primary Index
  - Data Skipping Index
  - Materialized View
  - Projection
  - Parallelism

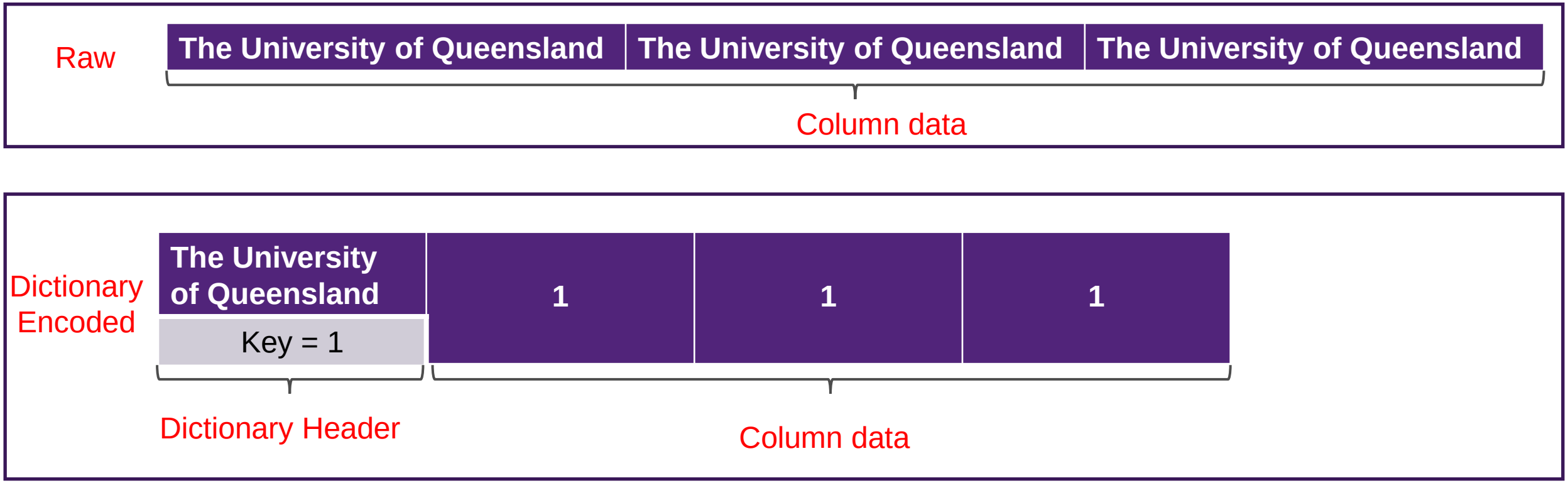
Columnar Store Compression

- Data stored in columns is more compressible than data stored in rows. Compression algorithms perform better on data with low information entropy (i.e., with high data value locality), and values from the same column tend to have more value locality than values from different columns.
- Column Encoding Techniques:
  - Dictionary Encoding
  - Run-Length Encoding
  - Delta Encoding
  - Bit-Packing Encoding



Dictionary Encoding

- Dictionary encoding works well for distributions with ***a few very frequent values***, and can also be applied to strings



Dictionary Encoding

- Dictionary encoding works well for distributions with ***a few very frequent values***, or strings
- The simplest form constructs a ***dictionary table*** for ***an entire table column sorted on frequency***, and represents values as the integer position in this table. These integers can again be compressed using an integer compression scheme.
- The global dictionary may grow large, and the value distribution may vary locally.
- Dictionary compression helps to optimize queries by **rewriting predicates on strings** into **predicates on integers**.
- Benefits:
  - ***fixed width columns*** if the system chooses all codes to be of the same width.
  - ***CPU efficient access*** patterns

Dictionary Encoding

```
SELECT name,
   formatReadableSize(sum(data_compressed_bytes)) AS compressed_size,
   formatReadableSize(sum(data_uncompressed_bytes)) AS uncompressed_size,
   round(sum(data_uncompressed_bytes) / sum(data_compressed_bytes), 2) AS ratio
FROM system.columns WHERE table = 'uk_price_default' GROUP BY name
```

- LowCardinality: Changes the internal representation of other data types to be dictionary-encoded.

```
CREATE TABLE uk_price_default(
price UInt32,
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-detached' = 2,
'detached' = 3, 'flat' = 4, 'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1, 'leasehold' = 2,
'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (postcode1, postcode2, addr1, addr2);
```

name	compressed_size	uncompressed_size	ratio
locality	19.72 MiB	189.77 MiB	9.62
type	10.59 MiB	27.94 MiB	2.64
county	4.58 MiB	363.18 MiB	79.36
duration	3.44 MiB	27.94 MiB	8.12
town	1.91 MiB	271.14 MiB	142.32
date	56.06 MiB	55.89 MiB	1
addr1	44.97 MiB	130.90 MiB	2.91
street	13.63 MiB	388.78 MiB	28.52
addr2	8.45 MiB	49.49 MiB	5.85
price	81.45 MiB	111.77 MiB	1.37
district	6.91 MiB	330.75 MiB	47.88
is_new	4.29 MiB	27.94 MiB	6.52
postcode2	6.28 MiB	111.64 MiB	17.78
postcode1	578.25 KiB	123.78 MiB	219.2

name	compressed_size	uncompressed_size	ratio
locality	13.49 MiB	55.29 MiB	4.1
type	10.59 MiB	27.94 MiB	2.64
county	1.41 MiB	28.00 MiB	19.87
duration	3.44 MiB	27.94 MiB	8.12
town	617.43 KiB	55.95 MiB	92.8
date	56.06 MiB	55.89 MiB	1
addr1	44.95 MiB	130.90 MiB	2.91
street	11.10 MiB	64.79 MiB	5.84
addr2	8.44 MiB	49.49 MiB	5.86
price	81.45 MiB	111.77 MiB	1.37
district	3.20 MiB	55.95 MiB	17.5
is_new	4.29 MiB	27.94 MiB	6.52
postcode2	6.19 MiB	55.91 MiB	9.03
postcode1	285.20 KiB	52.65 MiB	189.02

```
CREATE TABLE uk_price_dic(
price UInt32,
date Date,
postcode1 LowCardinality(String),
postcode2 LowCardinality(String),
type Enum8('terraced' = 1, 'semi-detached' = 2,
'detached' = 3, 'flat' = 4, 'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1, 'leasehold' = 2,
'unknown' = 0),
addr1 String,
addr2 String,
street LowCardinality(String),
locality LowCardinality(String),
town LowCardinality(String),
district LowCardinality(String),
county LowCardinality(String))
ENGINE = MergeTree
ORDER BY (postcode1, postcode2, addr1,
addr2);
```

Dictionary Encoding

- LowCardinality: Changes the internal representation of other data types to be dictionary-encoded.

```
CREATE TABLE uk_price_default(
price UInt32,
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-detached' = 2,
'detached' = 3, 'flat' = 4, 'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1, 'leasehold' = 2,
'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (postcode1, postcode2, addr1, addr2);
```

SELECT DISTINCT(postcode1) FROM uk_price_dic

```
2389 rows in set. Elapsed: 0.016 sec. Processed 29.30 million rows, 52.41 MB (1.
78 billion rows/s., 3.19 GB/s.)
Peak memory usage: 2.91 MiB.
```

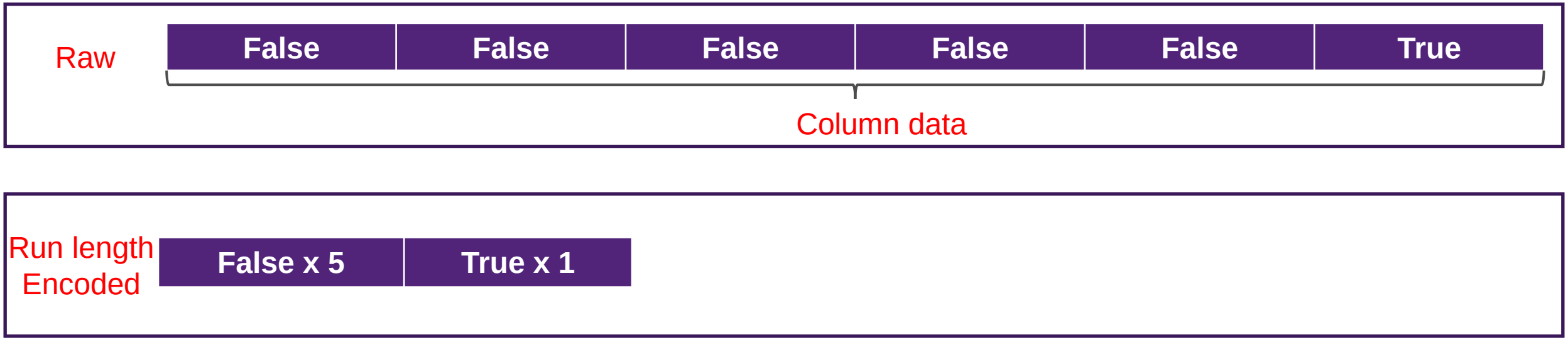
SELECT DISTINCT(postdoc1) FROM uk_price_default

```
2389 rows in set. Elapsed: 0.025 sec. Processed 29.30 million rows, 364.19 MB (1
.16 billion rows/s., 14.59 GB/s.)
Peak memory usage: 982.21 KiB.
```

```
CREATE TABLE uk_price_dic(
price UInt32,
date Date,
postcode1 LowCardinality(String),
postcode2 LowCardinality(String),
type Enum8('terraced' = 1, 'semi-detached' = 2,
'detached' = 3, 'flat' = 4, 'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1, 'leasehold' = 2,
'unknown' = 0),
addr1 String,
addr2 String,
street LowCardinality(String),
locality LowCardinality(String),
town LowCardinality(String),
district LowCardinality(String),
county LowCardinality(String))
ENGINE = MergeTree
ORDER BY (postcode1, postcode2, addr1,
addr2);
```

Run-Length Encoding

- Run-length encoding (RLE) compresses ***runs of the same value*** in a column to a compact singular representation.

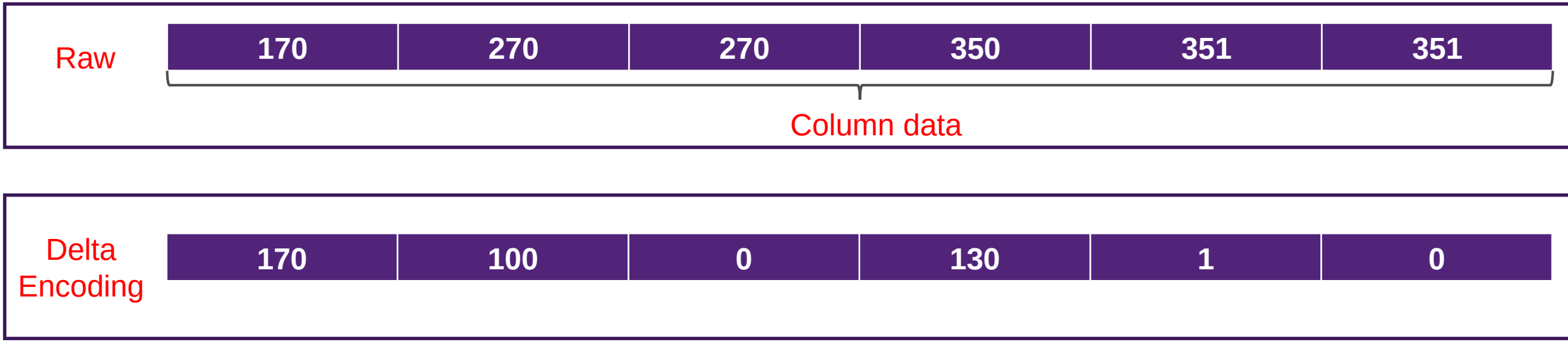


Run-Length Encoding

- Run-length encoding (RLE) compresses ***runs of the same value*** in a column to a compact singular representation.
- Well-suited for columns that are ***sorted*** or that have ***reasonable-sized runs of the same value***.
- These runs are replaced with triples: ***(value, start position, length)*** where each element of the triple is typically given a fixed number of bits.
  - ✓ For example, if the first 42 elements of a column contain the value ‘M’, then these 42 elements can be replaced with the triple: (‘M’, 1, 42).

Delta Encoding

- Storing the ***difference between consecutive values*** rather than the actual values. This method is particularly useful for compressing data with many consecutive values, such as timestamp data types.



Delta Encoding

- Storing the ***difference between consecutive values*** rather than the actual values. This method is particularly useful for compressing data with many consecutive values, such as timestamp data types.
  - 1. ***Initialize*** a variable to store the previous value: Initialize a variable to store the previous value, which will be used to calculate the delta.
  - 2. ***Traverse***: Traverse the data and for each value, calculate the delta by subtracting the previous value from the current value.
  - 3. ***Replace*** values with delta: Replace each value in the data with its corresponding delta.
  - 4. ***Store*** the initial value and the compressed data: Store the initial value and the compressed data (i.e the data with values replaced by deltas) separately.
  - 5. ***Decompression***: For decompression, use the stored initial value and the delta to calculate the original value.

Delta Encoding

```
SELECT name,
   formatReadableSize(sum(data_compressed_bytes)) AS compressed_size,
   formatReadableSize(sum(data_uncompressed_bytes)) AS uncompressed_size,
   round(sum(data_uncompressed_bytes) / sum(data_compressed_bytes), 2) AS ratio
FROM system.columns WHERE table = 'uk_price_default' GROUP BY name
```

- Delta Encoding works best for data that has many consecutive values.

```
CREATE TABLE uk_price_default(
price UInt32,
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-
detached' = 2, 'detached' = 3, 'flat' = 4,
'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1,
'leasehold' = 2, 'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (postcode1, postcode2,
addr1, addr2);
```

name	compressed_size	uncompressed_size	ratio
locality	19.72 MiB	189.77 MiB	9.62
type	10.59 MiB	27.94 MiB	2.64
county	4.58 MiB	363.18 MiB	79.36
duration	3.44 MiB	27.94 MiB	8.12
town	1.91 MiB	271.14 MiB	142.32
date	56.06 MiB	55.89 MiB	1
addr1	44.97 MiB	130.90 MiB	2.91
street	13.63 MiB	388.78 MiB	28.52
addr2	8.45 MiB	49.49 MiB	5.85
price	81.45 MiB	111.77 MiB	1.37
district	6.91 MiB	330.75 MiB	47.88
is_new	4.29 MiB	27.94 MiB	6.52
postcode2	6.28 MiB	111.64 MiB	17.78
postcode1	578.25 KiB	123.78 MiB	219.2

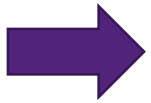
name	compressed_size	uncompressed_size	ratio
locality	58.73 MiB	189.77 MiB	3.23
type	17.81 MiB	27.94 MiB	1.57
county	28.54 MiB	363.18 MiB	12.73
duration	10.64 MiB	27.94 MiB	2.63
town	29.64 MiB	271.14 MiB	9.15
date	56.10 MiB	55.89 MiB	1
addr1	95.25 MiB	130.90 MiB	1.37
street	206.19 MiB	388.78 MiB	1.89
addr2	15.88 MiB	49.49 MiB	3.12
price	1.06 MiB	111.77 MiB	104.99
district	35.98 MiB	330.75 MiB	9.19
is_new	6.17 MiB	27.94 MiB	4.53
postcode2	78.35 MiB	111.64 MiB	1.42
postcode1	24.51 MiB	123.78 MiB	5.05

```
CREATE TABLE uk_price_DELTA(
price UInt32 CODEC(Delta, ZSTD),
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-
detached' = 2, 'detached' = 3, 'flat' = 4,
'other' = 0),
is_new UInt8 ,
duration Enum8('freehold' = 1,
'leasehold' = 2, 'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (price);
```

Delta Encoding

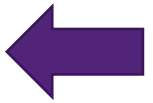
- Delta Encoding works best for data that has many consecutive values.

```
CREATE TABLE uk_price_default(
price UInt32,
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-
detached' = 2, 'detached' = 3, 'flat' = 4,
'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1,
'leasehold' = 2, 'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (postcode1, postcode2,
addr1, addr2);
```



```
SELECT sum(price)
FROM uk_price_default

Query id: e1537c4f-0993-4c40-846b-315c20d25162

    ┌─sum(price)─┐
1.  │ 6593749686057 │ -- 6.59 trillion
    └────────────┘

↗ Progress: 29.30 million rows, 117.20 MB (788.28 million rows/
↘ Progress: 29.30 million rows, 117.20 MB (788.28 million rows/

1 row in set. Elapsed: 0.038 sec. Processed 29.30 million rows,
 117.20 MB (772.29 million rows/s., 3.09 GB/s.)
Peak memory usage: 725.76 KiB.
```

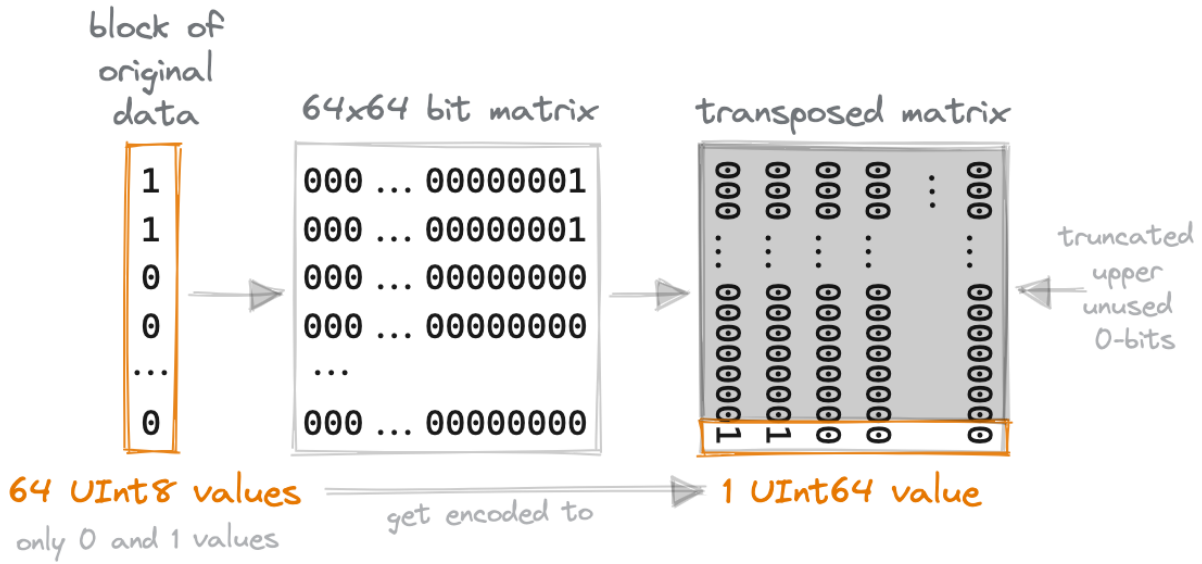
```
SELECT sum(price)
FROM uk_price_DELTA

Query id: bbdc0e4c-4052-4545-b62e-b26898cfa1c6

    ┌─sum(price)─┐
1.  │ 6593749686057 │ -- 6.59 trillion
    └────────────┘

↙ Progress: 29.30 million rows, 117.20 MB (2.61 billion rows/s.,
← Progress: 29.30 million rows, 117.20 MB (2.61 billion rows/s.,

1 row in set. Elapsed: 0.011 sec. Processed 29.30 million rows,
117.20 MB (2.56 billion rows/s., 10.25 GB/s.)
Peak memory usage: 365.85 KiB.
```

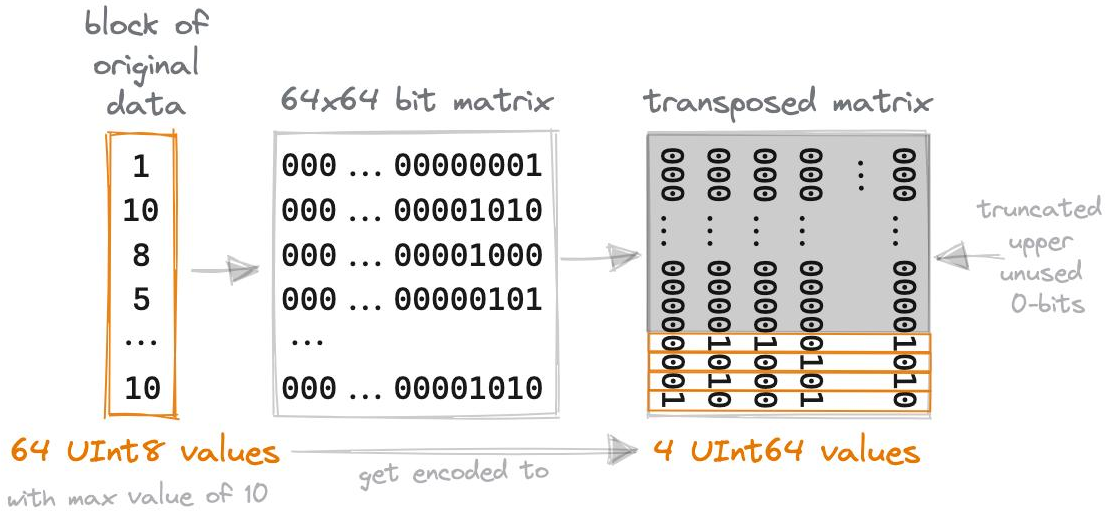


```
CREATE TABLE uk_price_DELTA(
price UInt32 CODEC(Delta, ZSTD),
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-
detached' = 2, 'detached' = 3, 'flat' = 4,
'other' = 0),
is_new UInt8 ,
duration Enum8('freehold' = 1,
'leasehold' = 2, 'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (price);
```

Bit-Packing Encoding – T64

- Finds the common high bits of 64 values and trims them, only recording the changing parts.
- Can be effective on sparse data or when the range in a block is small.
- Avoid T64 for random numbers.



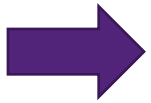


https://clickhouse.com/blog/optimize-clickhouse-codecs-compression-schema

Bit-Packing Encoding – T64

- CODEC(T64)

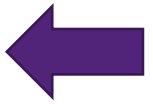
```
CREATE TABLE uk_price_default(
price UInt32,
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-
detached' = 2, 'detached' = 3, 'flat' = 4,
'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1,
'leasehold' = 2, 'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (postcode1, postcode2,
addr1, addr2);
```



name	compressed_size	uncompressed_size	ratio
locality	19.72 MiB	189.77 MiB	9.62
type	10.59 MiB	27.94 MiB	2.64
county	4.58 MiB	363.18 MiB	79.36
duration	3.44 MiB	27.94 MiB	8.12
town	1.91 MiB	271.14 MiB	142.32
date	56.06 MiB	55.89 MiB	1
addr1	44.97 MiB	130.90 MiB	2.91
street	13.63 MiB	388.78 MiB	28.52
addr2	8.45 MiB	49.49 MiB	5.85
price	81.45 MiB	111.77 MiB	1.37
district	6.91 MiB	330.75 MiB	47.88
is_new	4.29 MiB	27.94 MiB	6.52
postcode2	6.28 MiB	111.64 MiB	17.78
postcode1	578.25 KiB	123.78 MiB	219.2

name	compressed_size	uncompressed_size	ratio
locality	19.72 MiB	189.77 MiB	9.62
type	5.14 MiB	27.94 MiB	5.44
county	4.58 MiB	363.18 MiB	79.36
duration	3.44 MiB	27.94 MiB	8.12
town	1.91 MiB	271.14 MiB	142.31
date	56.07 MiB	55.89 MiB	1
addr1	44.97 MiB	130.90 MiB	2.91
street	13.63 MiB	388.78 MiB	28.52
addr2	8.45 MiB	49.49 MiB	5.85
price	81.45 MiB	111.77 MiB	1.37
district	6.91 MiB	330.75 MiB	47.88
is_new	4.29 MiB	27.94 MiB	6.52
postcode2	6.28 MiB	111.64 MiB	17.78
postcode1	578.68 KiB	123.78 MiB	219.03

```
SELECT name,
   formatReadableSize(sum(data_compressed_bytes)) AS compressed_size,
   formatReadableSize(sum(data_uncompressed_bytes)) AS uncompressed_size,
   round(sum(data_uncompressed_bytes) / sum(data_compressed_bytes), 2) AS ratio
FROM system.columns WHERE table = 'uk_price_default' GROUP BY name
```



```
CREATE TABLE uk_price_T64(
price UInt32,
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-
detached' = 2, 'detached' = 3, 'flat' = 4,
'other' = 0) CODEC(T64, ZSTD),
is_new UInt8,
duration Enum8('freehold' = 1,
'leasehold' = 2, 'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
ORDER BY (postcode1, postcode2,
addr1, addr2);
```

Outline

- Background and Concepts about Column-Oriented Databases
- ClickHouse - An open-source column-oriented DBMS
- Column-Oriented Database Compression
  - Dictionary Encoding
  - Run-length Encoding
  - Delta Encoding
  - Bit-Packing Encoding
- Optimizing Query Performance
  - Sparse Primary Index
  - Data Skipping Index
  - Materialized View
  - Projection
  - Parallelism

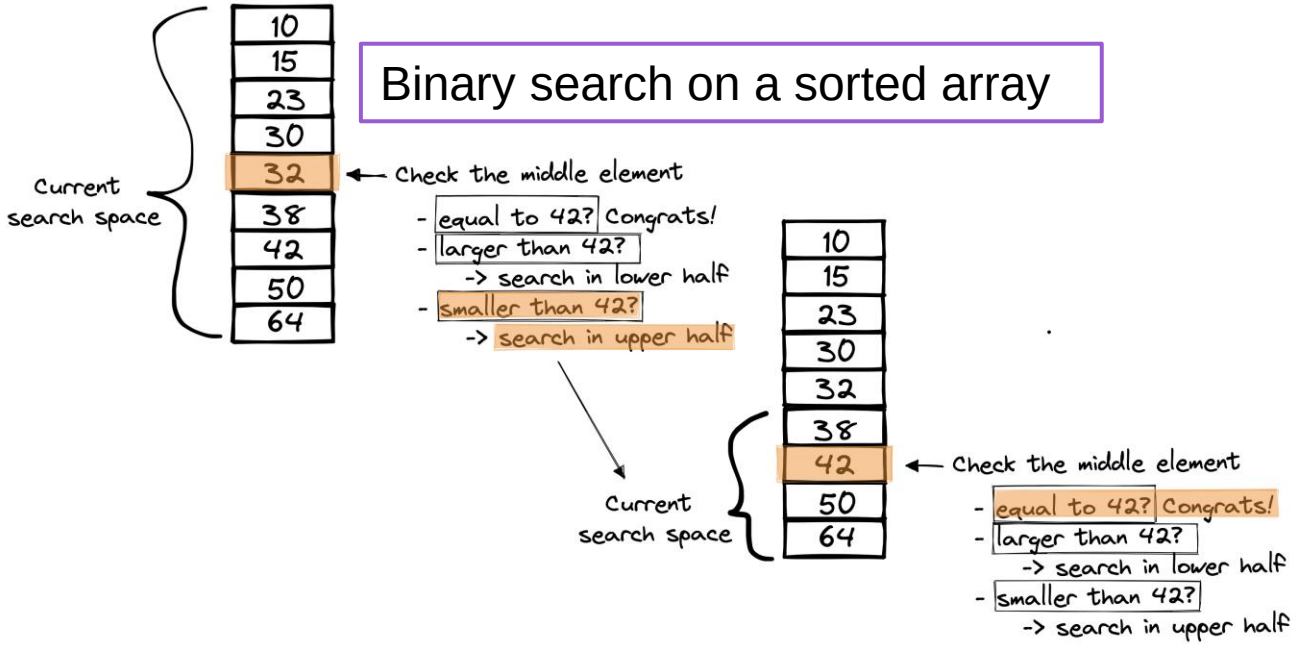
Optimizing Query Performance

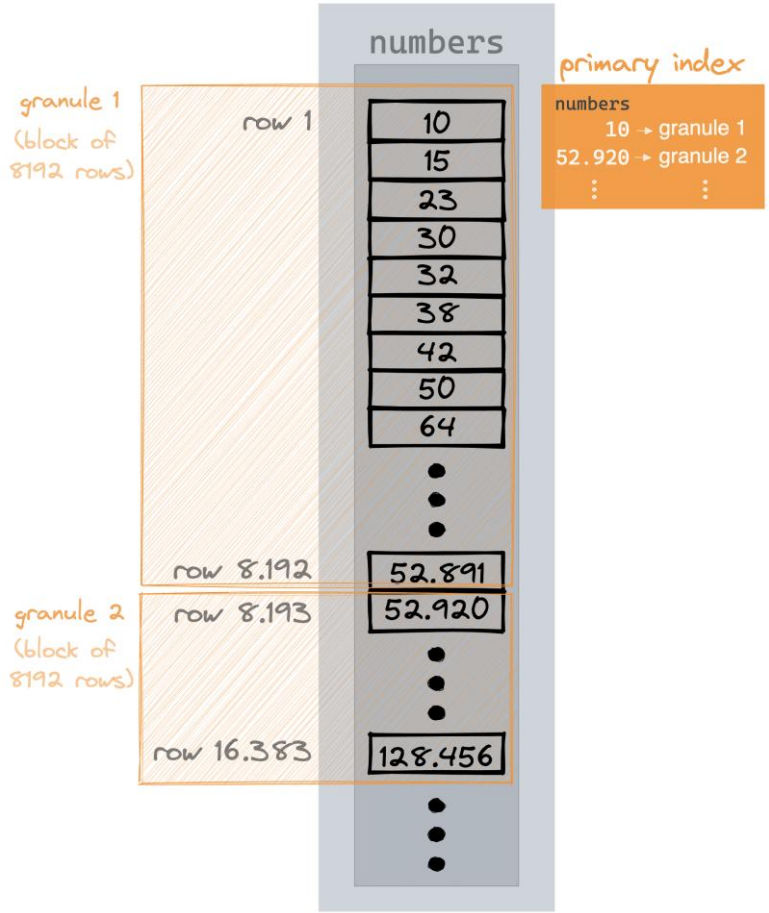
Query Optimization in column-oriented databases is different from traditional row-oriented database.

- **Sparse Primary Index**
  - limit the amount of data that needs to read from disk, prevent resorting of data at query time
- **Data Skipping Indexes**
  - Skip over irrelevant data parts while reading from disk
- **Materialized Views**
  - Materialized views allow users to shift the cost of computation from query time to insert time, resulting in faster SELECT queries.
- **Projections**
  - A projection is an additional (hidden) table that is automatically kept in sync with the original table.
- **Parallelism**
  - A column-oriented database server can process and insert data in parallel.

Sparse Primary Index

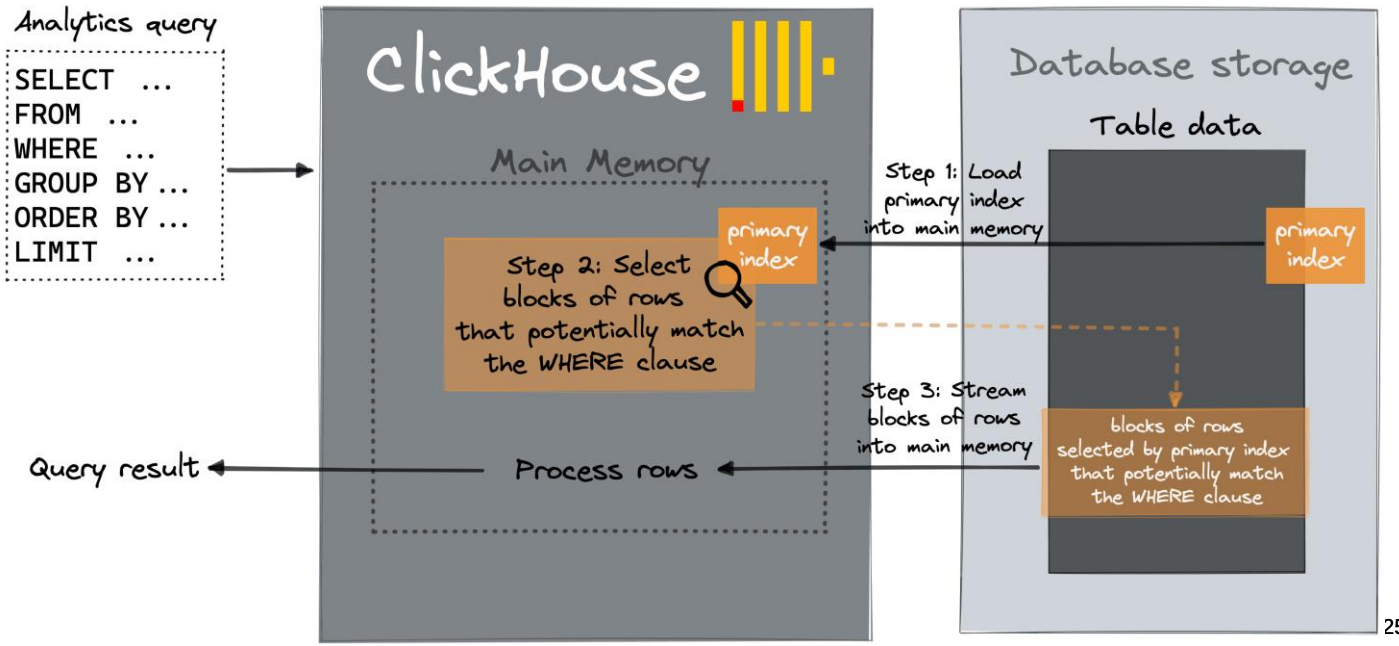
- ***Sparse primary index is used to limit the amount of data that needs to read from disk.***
- The design of the primary index is based on the ***binary search*** algorithm .
  - Efficiently finds the position of a target value within a sorted array.
  - Time complexity O(log n).
- The primary index is indexing blocks of rows.





Sparse Primary Index

- Step 1: The primary index from the involved table is loaded into the main memory.
- Step 2: Typically, via a binary search over the index entries, we selects blocks of rows that potentially contain rows matching the query's WHERE clause.
- Step 3: The selected blocks of rows are streamed in parallel into the query engine for further processing, and the query result is streamed to the caller.



25B

Sparse Primary Index

```
CREATE TABLE uk_price_prim(
price UInt32,
date Date,
postcode1 String,
postcode2 String,
type Enum8('terraced' = 1, 'semi-
detached' = 2, 'detached' = 3, 'flat' = 4,
'other' = 0),
is_new UInt8,
duration Enum8('freehold' = 1,
'leasehold' = 2, 'unknown' = 0),
addr1 String,
addr2 String,
street String,
locality String,
town String,
district String,
county String)
ENGINE = MergeTree
PRIMARY KEY (price)
ORDER BY (price)
SETTINGS index_granularity = 8192,
index_granularity_bytes = 0;
```

```
SELECT count(*)
FROM uk_price_default

Query id: f0bb04b1-95e4-4fe5-8391-ffedac1abf5e

    ┌─count()─┐
1.  │ 29299986 │ -- 29.30 million
    └─────────┘
```

SELECT price from uk_price_prim where price< 55000 or price > 200000

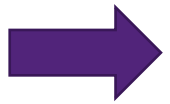
```
13753059 rows in set. Elapsed: 0.046 sec. Processed 13.83 million rows, 55.30 MB (297.81 mi
llion rows/s., 1.19 GB/s.)
Peak memory usage: 27.78 MiB.
```

Data Skipping Index

- ***Skip over irrelevant data*** parts while reading from disk
  - A type of secondary index that store ***summary information about primary key ranges***
  - When a query is executed, uses this summary information to determine which data parts need to be read, effectively reducing the amount of data that needs to be processed.
- Types
  - ***MinMax*** index: Stores the minimum and maximum values for a given column in each primary key range.
  - ***Set*** index: Stores a set of distinct values in a given column for each primary key range.
  - ***Bloom filter*** index: Stores a probabilistic data structure that allows for testing whether a given value is present in a column for each primary key range.
  - ***n-gram*** index: Stores information about n-grams (substrings of length n) for text columns in each primary key range.

Data Skipping Index

```
CREATE TABLE skip_table(
my_key UInt64,
my_value UInt64)
ENGINE MergeTree primary key my_key
SETTINGS index_granularity=8192;

INSERT INTO skip_table
SELECT number, intDiv(number,4096)
FROM numbers(100000000);
```



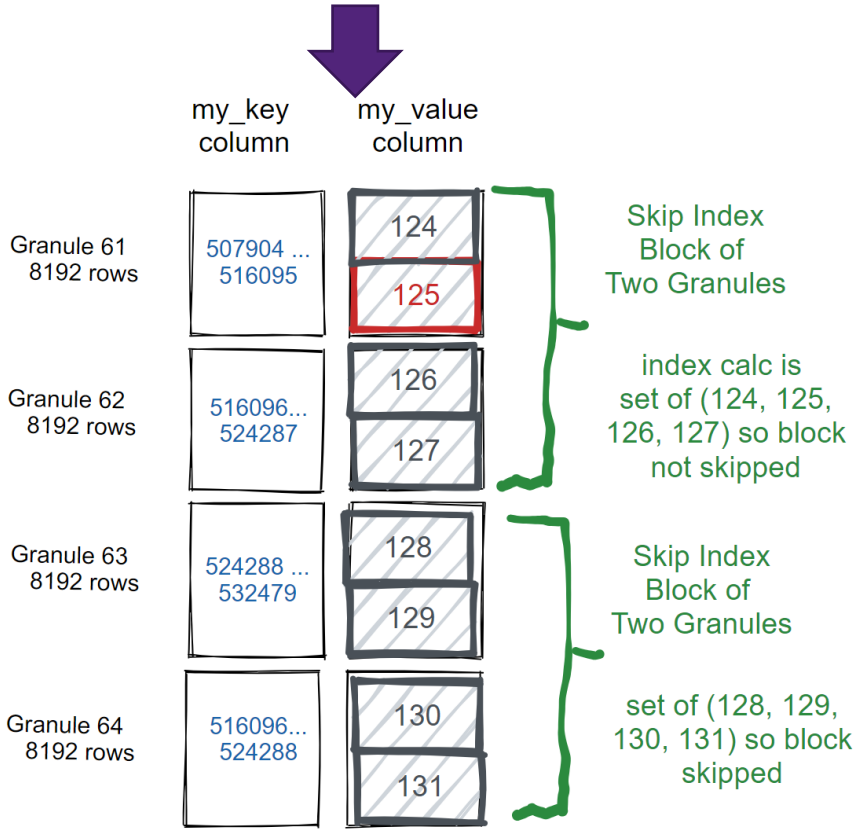
	my_key	my_value
1.	0	0
2.	1	0
3.	2	0
4.	3	0
5.	4	0
6.	5	0
7.	6	0
8.	7	0
9.	8	0
10.	9	0
11.	10	0
12.	11	0
13.	12	0
14.	13	0
15.	14	0

```
SELECT * FROM skip_table
WHERE my_value IN (125, 700)
```

```
8192 rows in set. Elapsed: 0.023 sec. Processed 100.00 million rows,
 800.10 MB (4.42 billion rows/s., 35.39 GB/s.)
Peak memory usage: 315.05 KiB.
```

Data Skipping Index

```
ALTER TABLE skip_table ADD INDEX vix my_value
TYPE set(100) GRANULARITY 2;
ALTER TABLE skip_table MATERIALIZE INDEX vix;
```

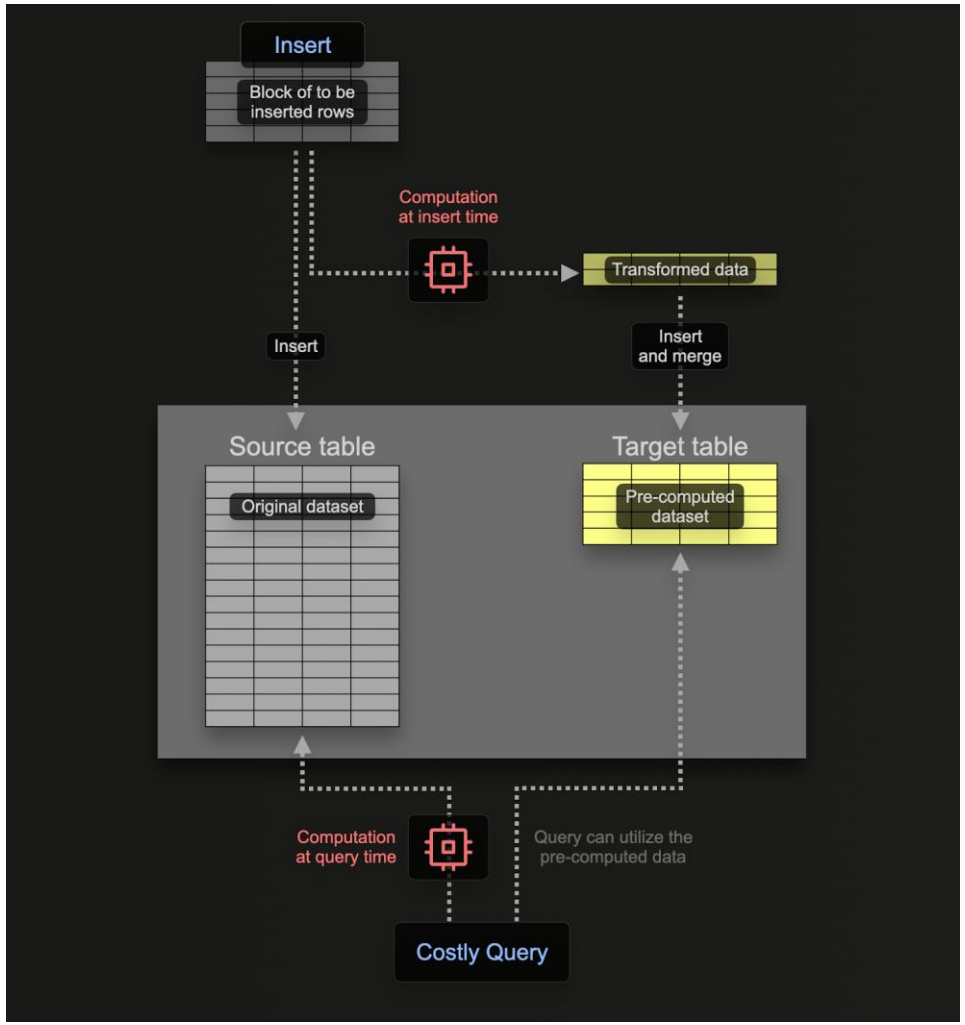



```
SELECT * FROM skip_table
WHERE my_value IN (125, 700)
```

```
8192 rows in set. Elapsed: 0.012 sec. Processed 32.77 thousand rows,
 360.45 KB (2.76 million rows/s., 30.36 MB/s.)
Peak memory usage: 190.48 KiB.
```

Materialized Views

- Materialized views allow users to shift the cost of computation from query time to insert time, resulting in faster SELECT queries.
- The result of this query is inserted into a second "target" table.
- Motivation for materialized views is that the results inserted into the target table represent the results of an aggregation, filtering, or transformation on rows.
- These results will often be a smaller representation of the original data.



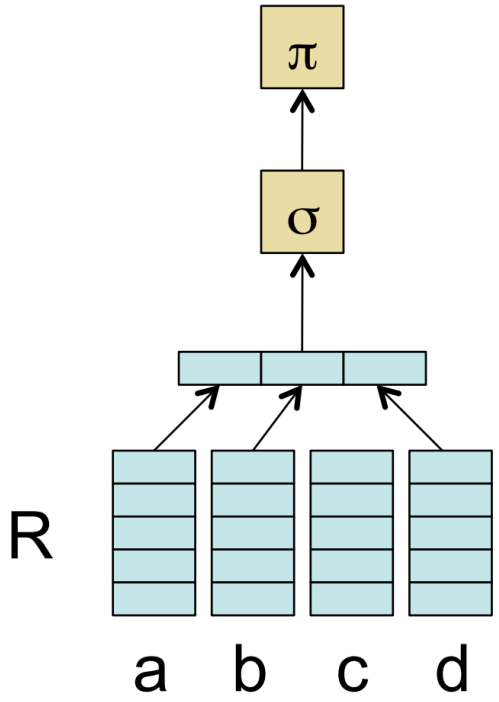
Early Materialization

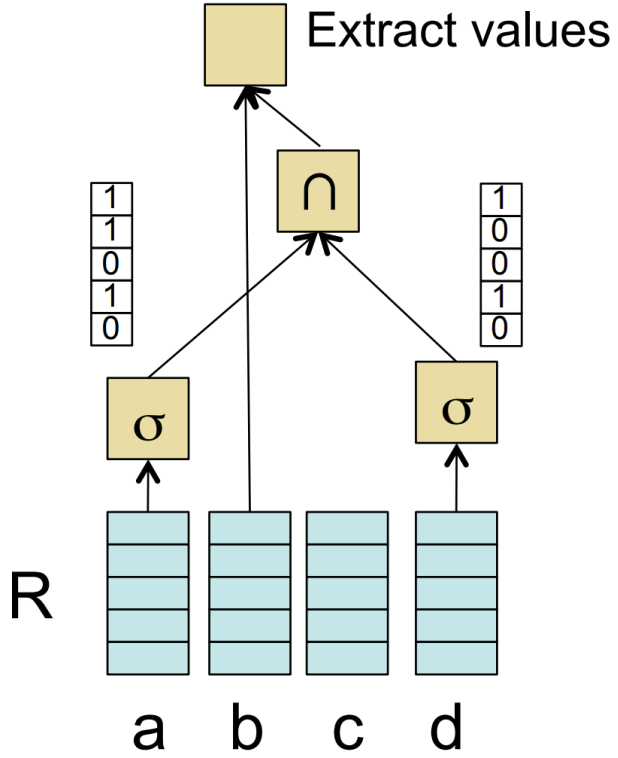
- Column-Oriented databases must bring data from various columns together into a single 'row' format to provide complete information about each entity.
- Traditional column-stores only ***load the relevant columns*** needed for a query from the disk or memory, combine them into rows ***early in the query process***, and then apply standard database operations like select, join, and aggregate.
- Although this approach works well, especially for analytical tasks in data warehousing, it doesn't fully capitalize on the performance advantages of storing data in columns.

Late Materialization

- More recent column-stores choose to ***keep data in columns until much later into the query plan***, operating directly on these columns. In order to do so, intermediate "position" lists often need to be constructed in order to match up operations that have been performed on different columns.

SELECT R.b from R where R.a=X and R.d=Y





Late Materialization Case Study

Initial Status

Relation R			Relation S	
Ra	Rb	Rc	Sa	Sb
3	12	12	17	11
16	34	34	49	35
56	75	53	58	62
9	45	23	99	44
11	49	78	64	29
27	58	65	37	78
8	97	33	53	19
41	75	21	61	81
19	42	29	32	26
35	55	0	50	23

Late Materialization Case Study

Initial Status

Relation R

Ra	Rb	Rc
3	12	12
16	34	34
56	75	53
9	45	23
11	49	78
27	58	65
8	97	33
41	75	21
19	42	29
35	55	0

Relation S

Sa	Sb
17	11
49	35
58	62
99	44
64	29
37	78
53	19
61	81
32	26
50	23

Query and Query Plan (MAL Algebra)

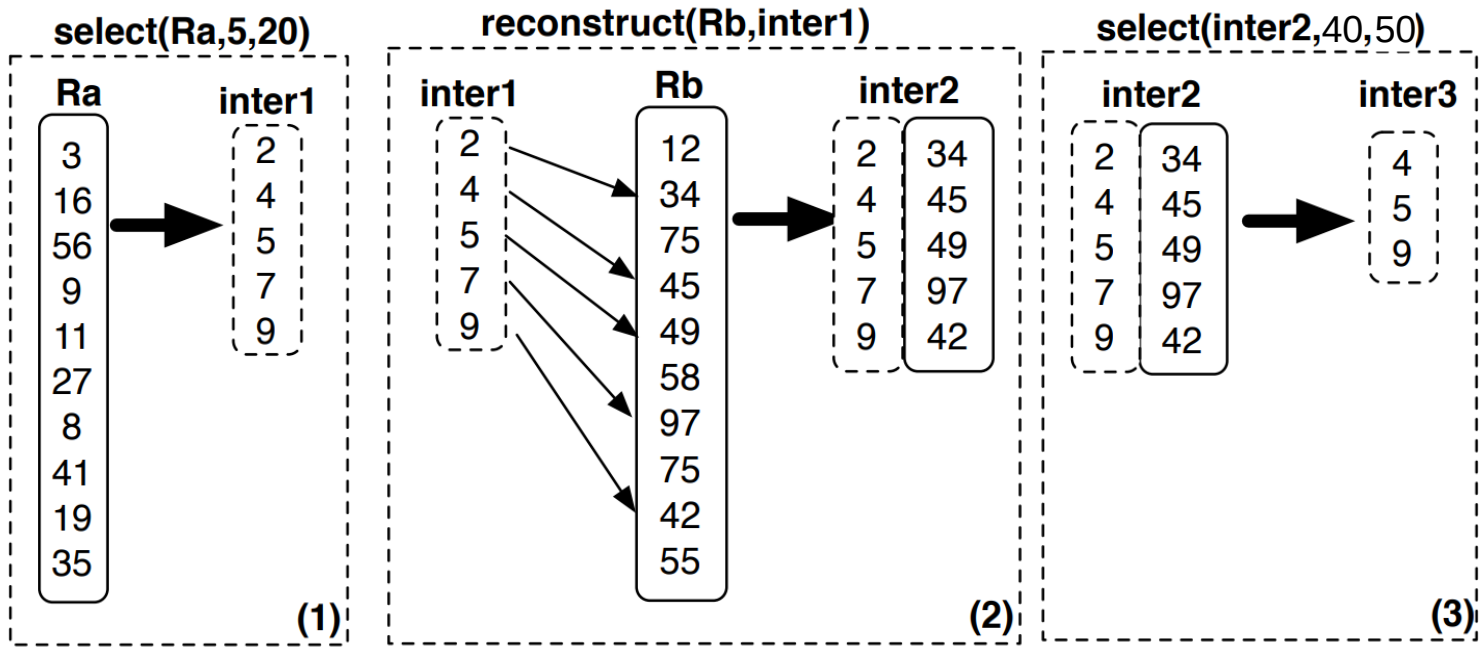
select sum(R.a) from R, S where R.c = S.b and
5<R.a<20 and 40<R.b<50 and 55<S.a<65

1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,40,50)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)

Late Materialization Case Study

select sum(R.a) from R, S where R.c = S.b and
5<R.a<20 and 40<R.b<50 and 55<S.a<65

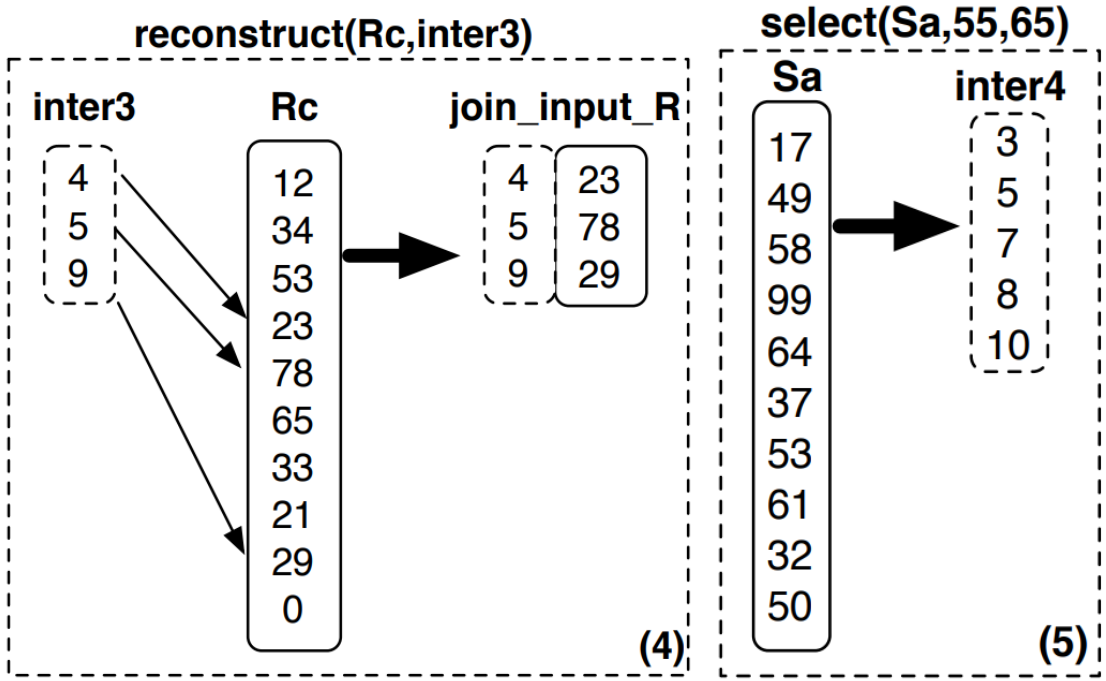
1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,40,50)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)



Late Materialization Case Study

select sum(R.a) from R, S where R.c = S.b and
5<R.a<20 and 40<R.b<50 and 55<S.a<65

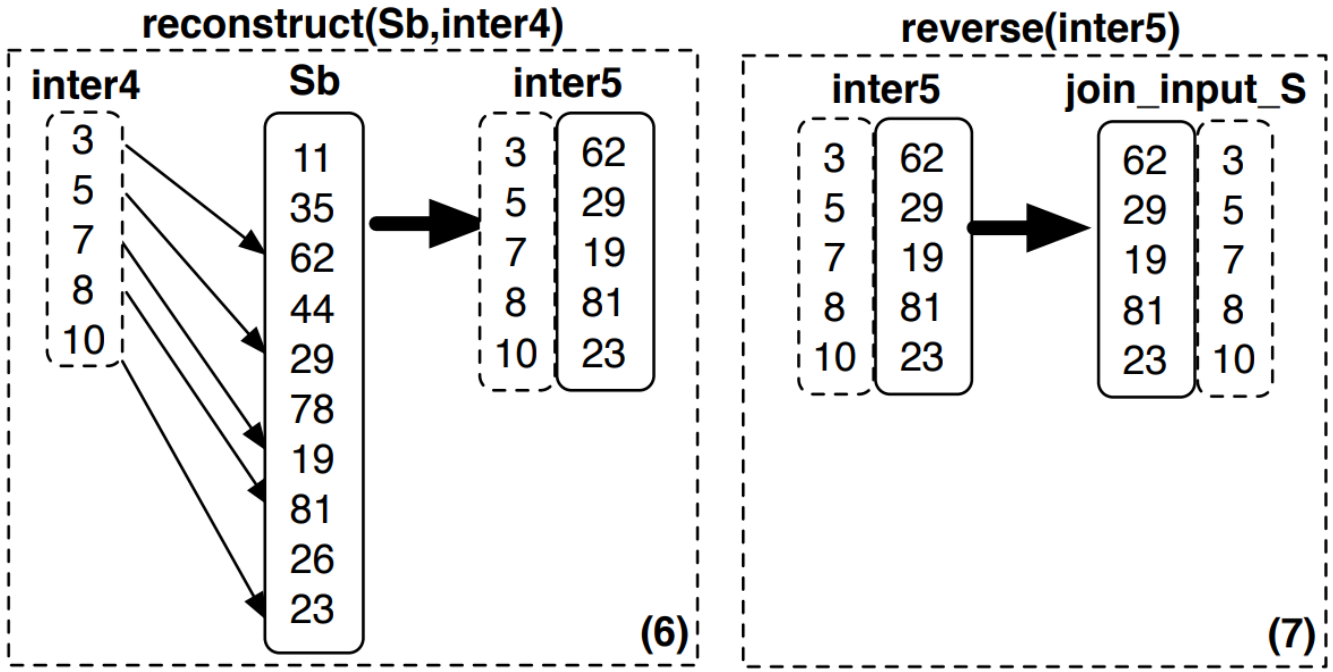
1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,40,50)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)



Late Materialization Case Study

select sum(R.a) from R, S where R.c = S.b and
5<R.a<20 and 40<R.b<50 and 55<S.a<65

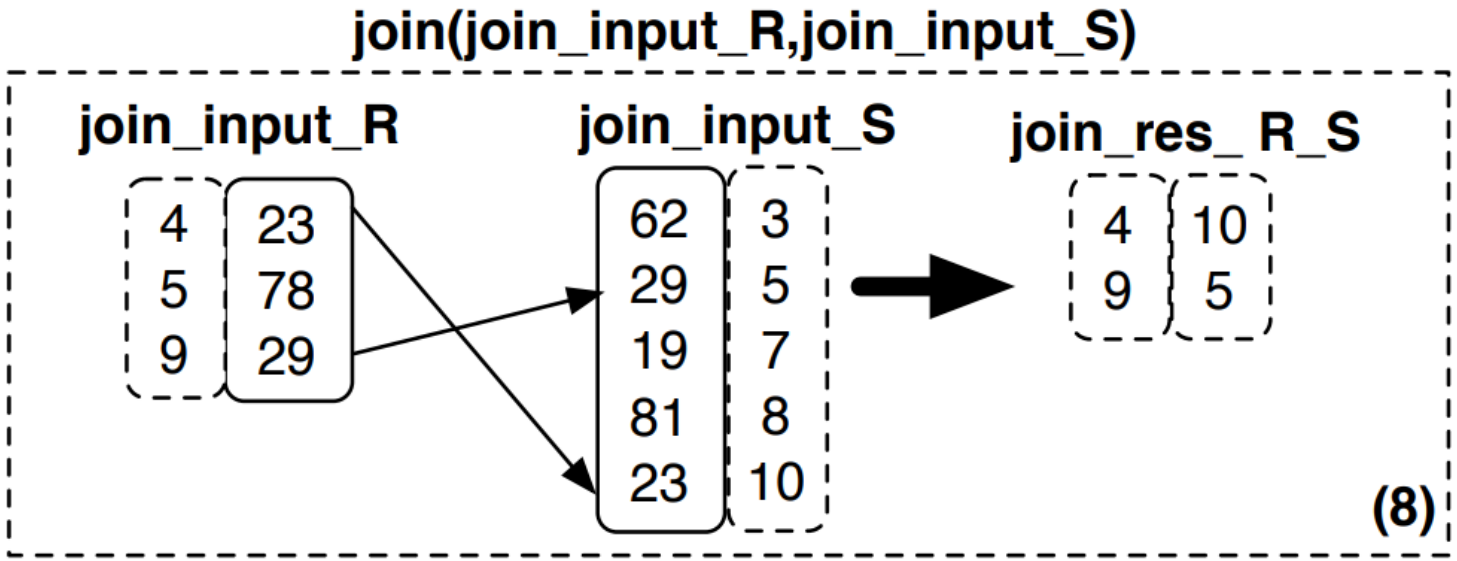
1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,40,50)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)



Late Materialization Case Study

select sum(R.a) from R, S where R.c = S.b and
5<R.a<20 and 40<R.b<50 and 55<S.a<65

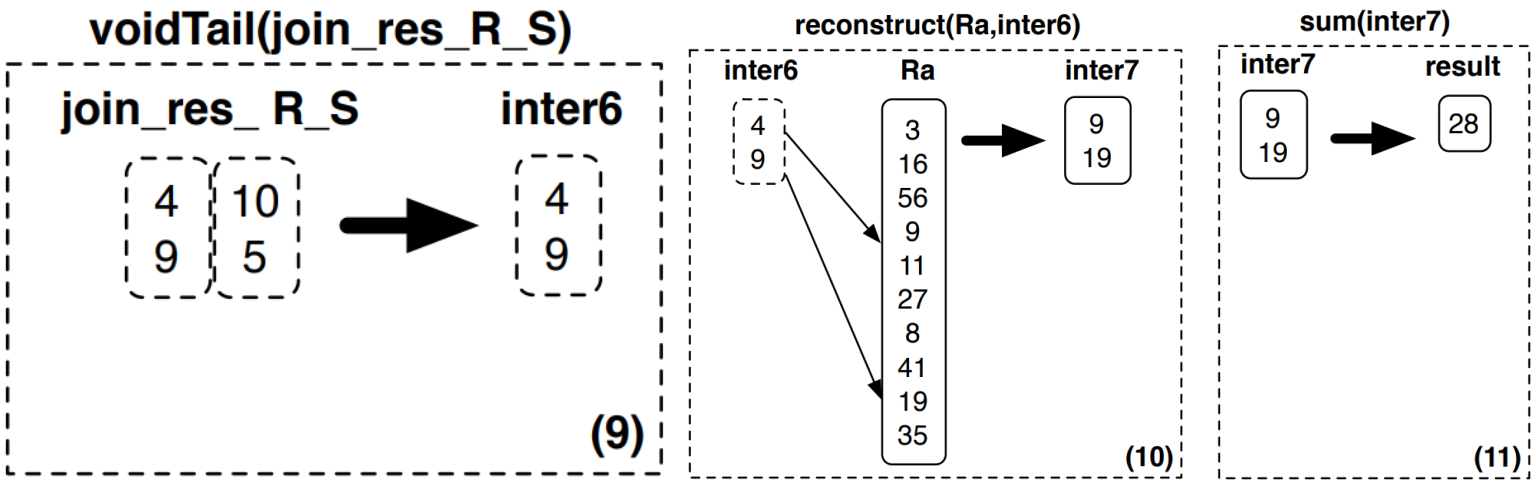
1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,40,50)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)



Late Materialization Case Study

select sum(R.a) from R, S where R.c = S.b and
5<R.a<20 and 40<R.b<50 and 55<S.a<65

1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,40,50)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)



Late Materialization Advantages

1. Reduced I/O Costs: Only the necessary columns required for a particular query are read from disk, reducing the amount of data loaded into memory. This is particularly beneficial for queries that access only a subset of columns from a wide table.

2. Improved Cache Utilization: By reading and processing only the needed columns, late materialization makes better use of the cache hierarchy in modern processors. It minimizes cache misses and increases the efficiency of data processing.

3. Lower Memory Overhead: As data from irrelevant columns isn't combined at the outset, the memory overhead is significantly reduced. This is crucial for handling large datasets where conserving memory can lead to performance improvements.

4. Enhanced Query Performance: Late materialization allows the database engine to optimize query execution plans based on the actual data access patterns observed during query execution. It can dynamically decide when to combine column data based on the query's requirements.

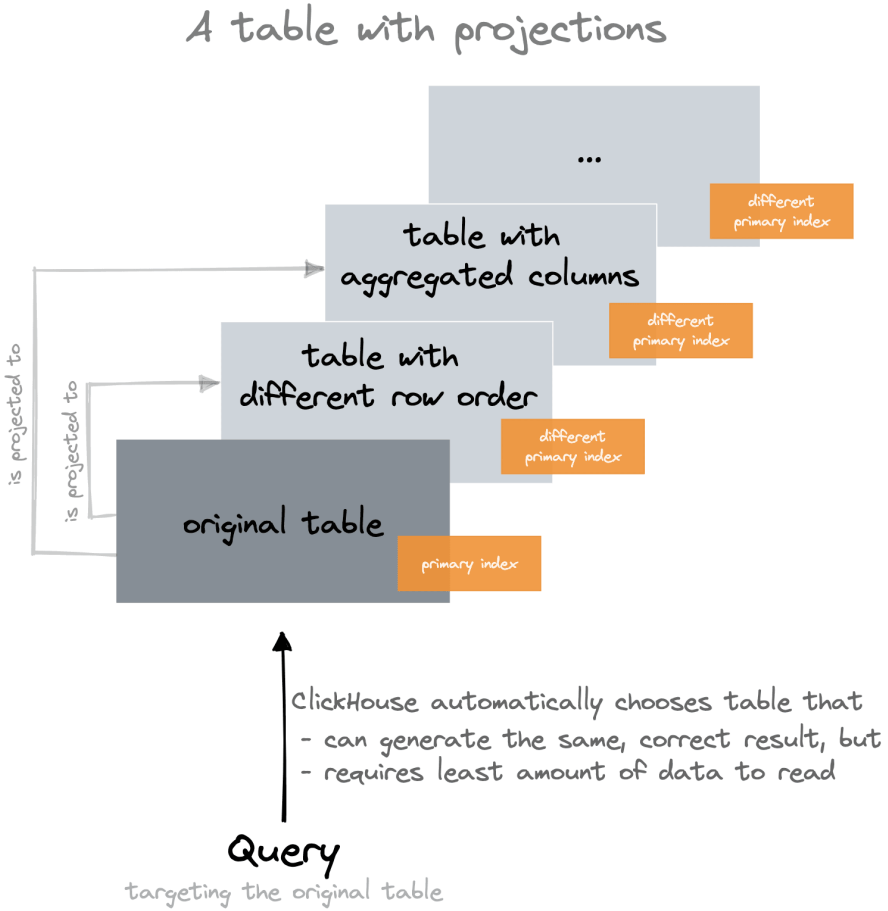
5. Flexibility in Query Execution: Queries can be adapted on the fly based on intermediate results. For instance, if an early part of the query filters out a large portion of data, subsequent operations will need to process significantly less data, hence speeding up the overall query execution.

6. Facilitates Vectorized Execution: Since data is maintained in columns, operations on the data can be vectorized more effectively. Processors can execute operations on entire columns at once, improving the speed of batch processing and analytical queries.

7. Better for Sparse Columns: In scenarios with many null or sparse values, late materialization avoids unnecessary combination and manipulation of empty data sets, which is often the case in columns with high sparsity.

Projections

- A projection is an additional (hidden) table that is automatically kept in sync with the original table.
- The projection can have a ***different row order*** (and therefore a ***different primary index***) than the original table, as well as automatically and incrementally pre-compute aggregate values.
- When a query is targeting the original table, then automatically chooses (by sampling the primary keys) a table that can generate the same correct result, but requires the least amount of data to be read.



Projection - Demo

Let's create a table to demonstrate the power of projection.

```
CREATE TABLE video_log (
`datetime` DateTime, -- 20,000 records per second
`user_id` UInt64, -- Cardinality == 100,000,000
`device_id` UInt64, -- Cardinality == 200,000,000
`domain` LowCardinality(String), -- Cardinality == 100
`bytes` UInt64, -- Ranging from 128 to 1152
`duration` UInt64 -- Ranging from 100 to 400
)
ENGINE = MergeTree
PARTITION BY toDate(datetime) -- Daily partitioning
ORDER BY (user_id, device_id); -- Can only favor one column here
```

Let's populate the table with one day's data using a GenerateRandom table.

```
CREATE TABLE rng (
 `user_id_raw` UInt64,
`device_id_raw` UInt64,
`domain_raw` UInt64, `bytes_raw` UInt64,
`duration_raw` UInt64 )
ENGINE = GenerateRandom(1024);
```

```
INSERT INTO video_log
SELECT
toUnixTimestamp(toDateTime(today())) + (rowNumberInAllBlocks() / 20000),
user_id_raw % 100000000 AS user_id,
device_id_raw % 200000000 AS device_id,
domain_raw % 100,
(bytes_raw % 1024) + 128,
(duration_raw % 300) + 100
FROM rng
LIMIT 172800000;
```

Projection - Demo

Case 1: Finding the hourly video stream property of a given user today.

```
SELECT
toStartOfHour(datetime) AS hour,
sum(bytes),
avg(duration)
FROM video_log
WHERE (toDate(hour) = today()) AND (user_id = 100)
GROUP BY hour;
```

```
   ┌────────────────hour─┬─sum(bytes)─┬─avg(duration)─┐
1. │ 2024-08-24 01:00:00 │       1011 │           172 │
   └─────────────────────┴────────────┴───────────────┘

↑ Progress: 73.73 thousand rows, 885.23 KB (5.35 million rows/s., 64.26 MB/s.)
↗ Progress: 73.73 thousand rows, 885.23 KB (5.35 million rows/s., 64.26 MB/s.)

1 row in set. Elapsed: 0.014 sec. Processed 73.73 thousand rows, 885.23 KB (5.25
 million rows/s., 63.06 MB/s.)
Peak memory usage: 127.91 KiB.
```

Projection - Demo

Case 2: Finding the hourly video stream property of a given device today.

```
SELECT
 toStartOfHour(datetime) AS hour,
 sum(bytes),
 avg(duration)
FROM video_log
WHERE (toDate(hour) = today()) AND (device_id = '100')
GROUP BY hour;
```

```
↙ Progress: 92.09 million rows, 1.11 GB (728.59 million rows/s., 8.74 GB/s.)  52
← Progress: 92.09 million rows, 1.11 GB (728.59 million rows/s., 8.74 GB/s.)  52
                    ┌────────────────hour─┬─sum(bytes)─┬─avg(duration)─┐
1.                  │ 2024-08-24 01:00:00 │        820 │           227 │
                    └─────────────────────┴────────────┴───────────────┘
↖ Progress: 92.09 million rows, 1.11 GB (728.59 million rows/s., 8.74 GB/s.)  52
↑ Progress: 172.80 million rows, 2.07 GB (799.86 million rows/s., 9.60 GB/s.)  9
↗ Progress: 172.80 million rows, 2.07 GB (799.86 million rows/s., 9.60 GB/s.)  9

1 row in set. Elapsed: 0.216 sec. Processed 172.80 million rows, 2.07 GB (799.05
 million rows/s., 9.59 GB/s.)
Peak memory usage: 3.85 MiB.
```

Projection - Demo

Case 2: Finding the hourly video stream property of a given device today.

Let's add a normal projection : p_norm to speed up querying by device_id.

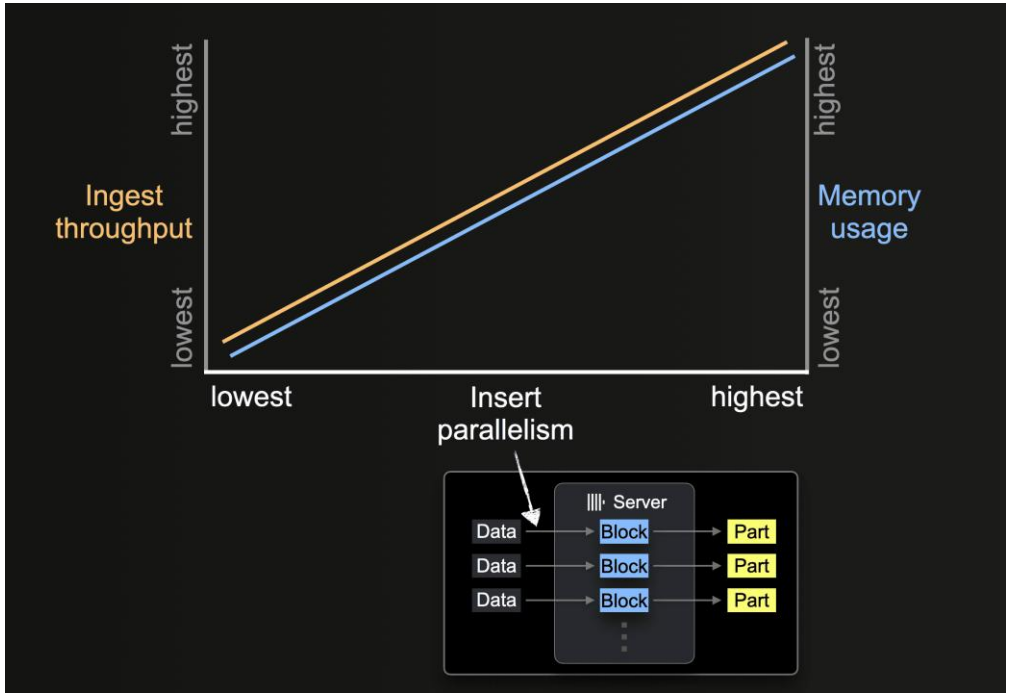
```
ALTER TABLE video_log ADD PROJECTION p_norm (
 SELECT
 datetime,
 device_id,
 bytes,
 duration
 ORDER BY device_id );
ALTER TABLE video_log MATERIALIZE PROJECTION p_norm;
```

```
SELECT
 toStartOfHour(datetime) AS hour,
 sum(bytes),
 avg(duration)
FROM video_log
WHERE (toDate(hour) = today())
AND (device_id = '100')
GROUP BY hour;
```

```
    ┌────────────────hour─┬─sum(bytes)─┬─avg(duration)─┐
1.  │ 2024-08-24 01:00:00 │        820 │           227 │
    └─────────────────────┴────────────┴───────────────┘

↘ Progress: 57.34 thousand rows, 1.61 MB (4.29 million rows/s., 120.20 MB/s.)  9
↓ Progress: 57.34 thousand rows, 1.61 MB (4.29 million rows/s., 120.20 MB/s.)  9

1 row in set. Elapsed: 0.014 sec. Processed 57.34 thousand rows, 1.61 MB (4.22 m
illion rows/s., 118.18 MB/s.)
Peak memory usage: 684.10 KiB.
```

Parallelism

- A column-oriented database server can process and insert data in parallel.
- The level of insert parallelism impacts the ingest throughput and memory usage of server.
- Loading and processing data in parallel requires more main memory but increases the ingest throughput as data is processed faster.



When to use ClickHouse?

- Don’t need ***low-latency stream ingestion***, and can afford to ***wait for micro-batching***
- Have ***few data sources*** that never (or rarely) change
- Never (or rarely) need to ***scale your cluster up or down*** to adjust to changing needs
- Can mix all queries into a single pool without ***differentiating priorities***
- Need a more ***simple environment***

What if my application does not fit into above situations, but I still want to keep the nice properties of column-oriented databases?

- Want ***automatically managed indexes***
- Have many data sources or schemas that change, and want ***automated management***
- Want to JOIN ***different data sources*** or use SQL transformation as part of data ingestion
- Need to ensure that ***high-priority queries*** aren't slowed by lower-priority queries
- Require a mix of high performance for recent and frequently-accessed data with lower costs for older and less accessed data

Druid – At a Glance

druid

- **Interactive conversations with data**
  - allowing humans and, sometimes, machines, to quickly and easily see and comprehend complex information
- **High concurrency**
  - large numbers of users, thousands or more, each generating multiple queries as they interact with the data.
- **Combine both real-time and historical data**
  - Data from past streams and from other sources, such as transactional systems, is delivered as a batch, through extract-load-transformation processes.

Druid vs ClickHouse

	druid	ClickHouse
Indexing	**Auto-index** applies the best index to each column	User **must choose** and manage indexes for each column
Ingestion	Both **batch and stream** ingestion. Events in streams are immediately available for query. Schema for incoming data is auto-detected. If new data has a different schema, the table can be automatically adjusted to fit the data.	**Only batch** ingestion. Streams are ingested through micro-batching, with a short delay before availability. Schema for every table must be explicitly defined before data is ingested. If new data has a different schema, ingestion fails.
Concurrency	Supports **hundreds to thousands** of concurrent queries.	Recommended **limit of 200 concurrent** queries
Mixed Workloads	Query queuing and service tiering to assign resources to queries based on **priority**.	**All queries are in one pool**, so can only assign priority by creating multiple ClickHouse clusters
Backups	All data **continuously backed up** into Deep Storage. Zero data loss during system outages.	**Backups must be scheduled and managed**. In an outage, all data since the previous backup can be lost.

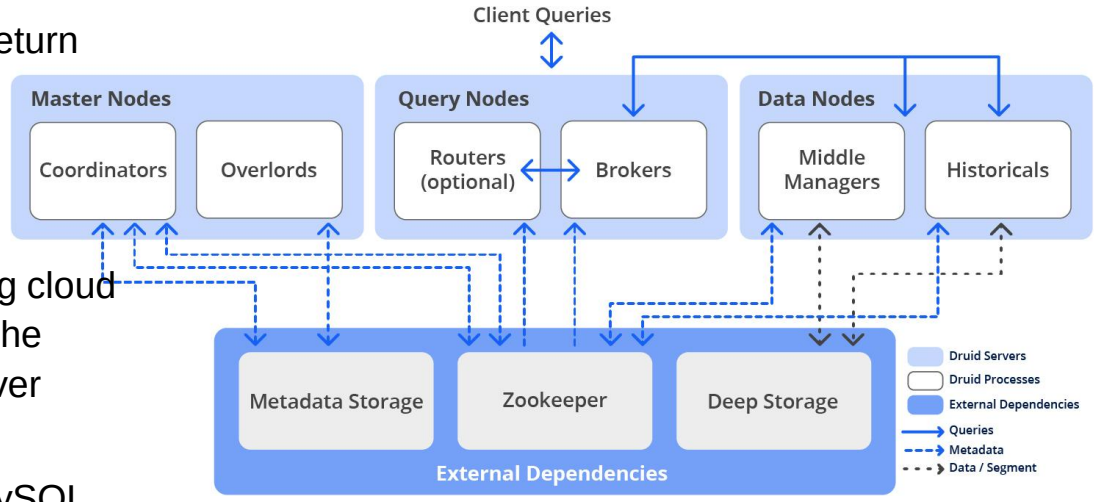
Druid Foundation of Design

Druid processes are deployed on **three server nodes**:

- **Master Nodes** govern data availability and ingestion
- **Query Nodes** accept queries, execute them across the system, and return the results
- **Data Nodes** execute ingestion workloads and store queryable data.

Three external dependencies:

- **Deep Storage** contains an additional copy of each data segment using cloud storage or HDFS. It allows data to move between Druid processes in the background and also provides a highly durable source of data to recover from system failures.
- **Metadata Storage** is a small relational database, Apache Derby™, MySQL, or PostgreSQL, that holds data about managing the data. It is used by Druid processes to keep track of storage segments, active tasks, and other configuration information.
- **Apache ZooKeeper** is primarily used for service discovery and leader election. It was historically used for other purposes as well, but they have been migrated away.



Summary of Column-Oriented Databases

- OLTP and OLAP database management systems are optimized for different kinds of workloads
- Columnar-oriented OLAP databases provide high-performance aggregation query execution capabilities on huge volumes of data
- Columnar compression and data indexing capabilities are available to build optimized analytics solutions

References

1. Nigel Pendse (August 23, 2007). "The origins of today's OLAP products". OLAP Report. Archived from the original on December 21, 2007. Retrieved November 27, 2007.
2. Abadi, Daniel J., Samuel R. Madden, and Nabil Hachem. "Column-stores vs. row-stores: how different are they really?." Proceedings of the 2008 ACM SIGMOD international conference on Management of data. 2008.
3. https://www.tinybird.co/blog-posts/what-is-a-columnar-database
4. http://daslab.seas.harvard.edu/classes/cs165/
5. https://stratos.seas.harvard.edu/files/stratos/files/columnstoresfntdbs.pdf
6. https://roundup.getdbt.com/p/how-does-column-encoding-work
7. https://docs.cloudera.com/cdw-runtime/cloud/impala-reference/topics/impala-lazy-materialization.html
8. https://courses.cs.washington.edu/courses/cse544/09wi/lecture-notes/lecture16/lecture16.pdf
9. https://imply.io/druid-architecture-concepts/
10. https://imply.io/blog/apache-druid-vs-clickhouse/
11. https://presentations.clickhouse.com/percona2021/projections.pdf