

# Copyright Notice

These slides are distributed under the Creative Commons License.

[DeepLearning.AI](#) makes these slides available for educational purposes. You may not use or distribute these slides for commercial purposes. You may make copies of these slides and use or distribute them for educational purposes as long as you cite [DeepLearning.AI](#) as the source of the slides.

For the rest of the details of the license, see <https://creativecommons.org/licenses/by-sa/2.0/legalcode>



DeepLearning.AI

# Data Storage and Queries

---

## Queries



DeepLearning.AI

# Queries

---

## **Week 3 Overview**

# What is a Query?

## Query

A statement that you write in a specific query language to retrieve or act on data.

Course 2 Lab



Relational Database Management System

SQL queries



```
%%sql
UPDATE category_copy
SET last_update = '2020-09-12 08:00:00.000';

UPDATE category_copy
SET category_id = '2'
WHERE name = 'Animation';
```

```
%%sql
SELECT *
FROM category_copy;
```

```
%%sql
INSERT INTO category_copy
VALUES
('1', 'Horror', '2006-02-15 09:46:27.000'),
('10', 'Animation', '2006-02-15 09:46:27.000'),
('20', 'Pop', '2006-02-15 09:46:27.000');
```

# What is a Query?

## Query

A statement that you write in a specific query language to retrieve or act on data.

Course 2 Lab



Amazon Object  
Storage

SQL-Like queries



```
file_s3_key = 'csv/ratings_ml_training_dataset.csv'
kwargs = {'ExpressionType': 'SQL',
          'Expression': """SELECT * FROM s3object s WHERE s.\"productline\" = 'Trains' LIMIT 20""",
          'InputSerialization': {'CSV': {'FileHeaderInfo': 'Use'}, 'CompressionType': 'NONE'},
          'OutputSerialization': {'CSV': {}}},

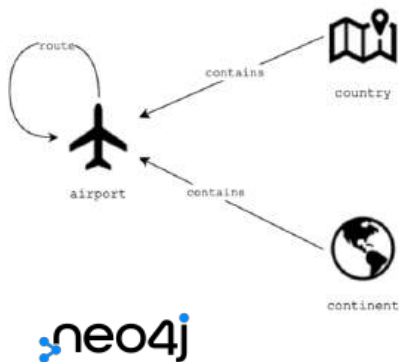
response = s3_select_object_content(bucket_name=BUCKET_NAME, object_key=file_s3_key, **kwargs)
```

# What is a Query?

## Query

A statement that you write in a specific query language to retrieve or act on data.

Course 3 Week 1 Lab



Cypher queries

```
query = """  
    MATCH (a:Airport)-[:Route]-(b:Airport)  
    RETURN a,b  
    LIMIT 10  
    """  
records = execute_query(query)  
print(records)
```

```
query = "MATCH ()-[r:Route]->() RETURN avg(r.dist)"  
records = execute_query(query)  
print(records)
```

# What is a Query?

## Query

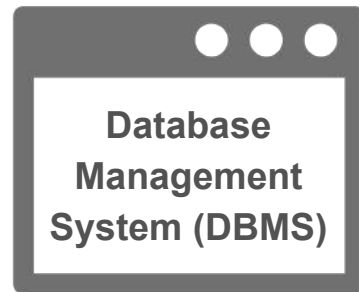
A statement that you write in a specific query language to retrieve or act on data.

---

## Query Languages

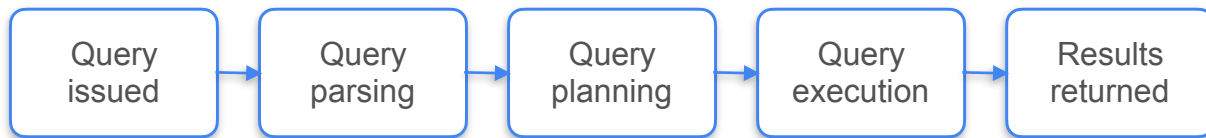
### Declarative language

- Your queries describe what data you want to retrieve
- Execution steps are abstracted from you and handled by the DBMS



# Week 3 Plan

- The journey of a query



- Techniques to improve SQL query performance (eg. database index)
- Many SQL query techniques are applicable to other query languages
- Aggregating queries: columnar versus row storage
- Queries on streaming data

---

## Labs

- Advanced SQL statements
- Execution time of an analytical query on row versus columnar storage
- Time-based windowed query on streaming data



Amazon Managed Service  
for Apache Flink





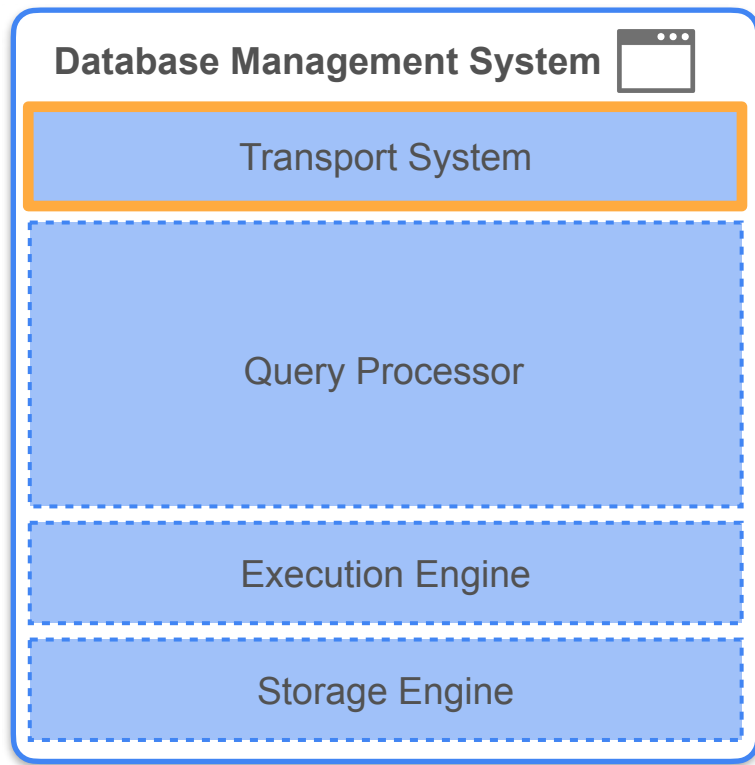
DeepLearning.AI

# Queries

---

## **The Life of a Query**

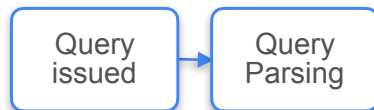
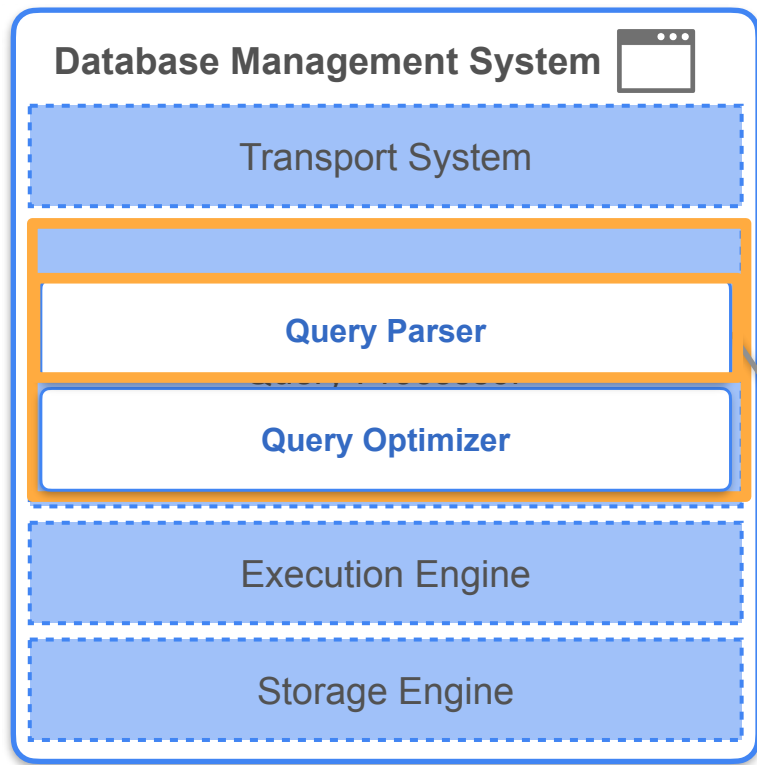
# Life of a Query



Query  
issued

```
SELECT * FROM customer;
```

# Life of a Query



0384fa1b059c



Breaks down the query into query tokens



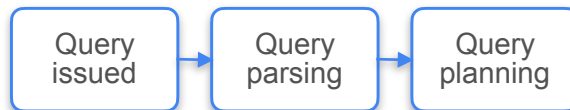
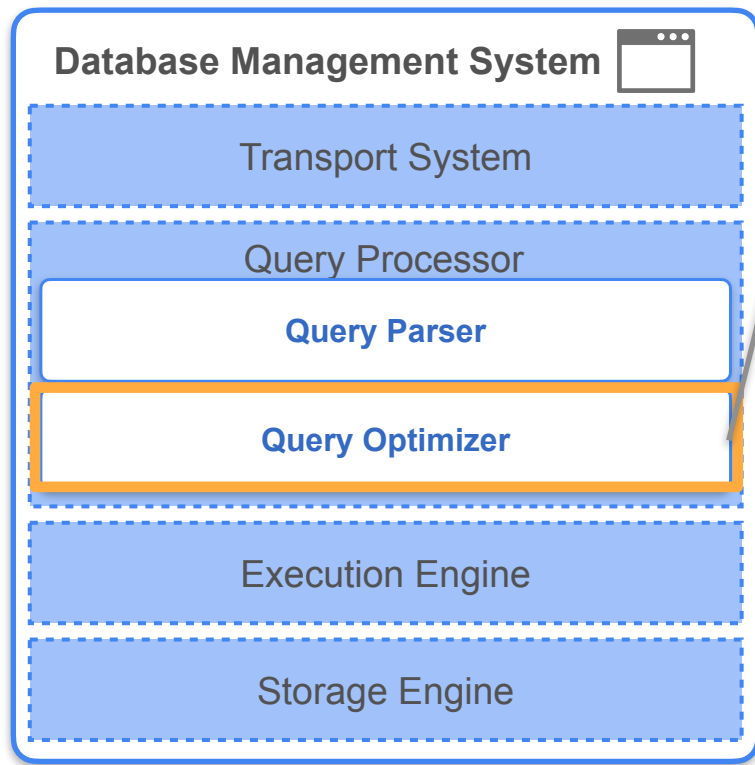
Checks proper syntax and validates the query



Performs control checks

0101 Converts into bytecode

# Life of a Query



Devises an execution plan:

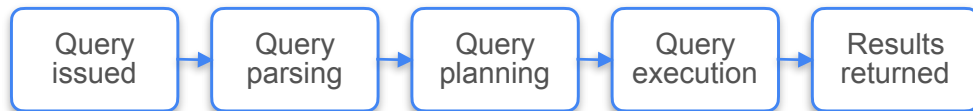
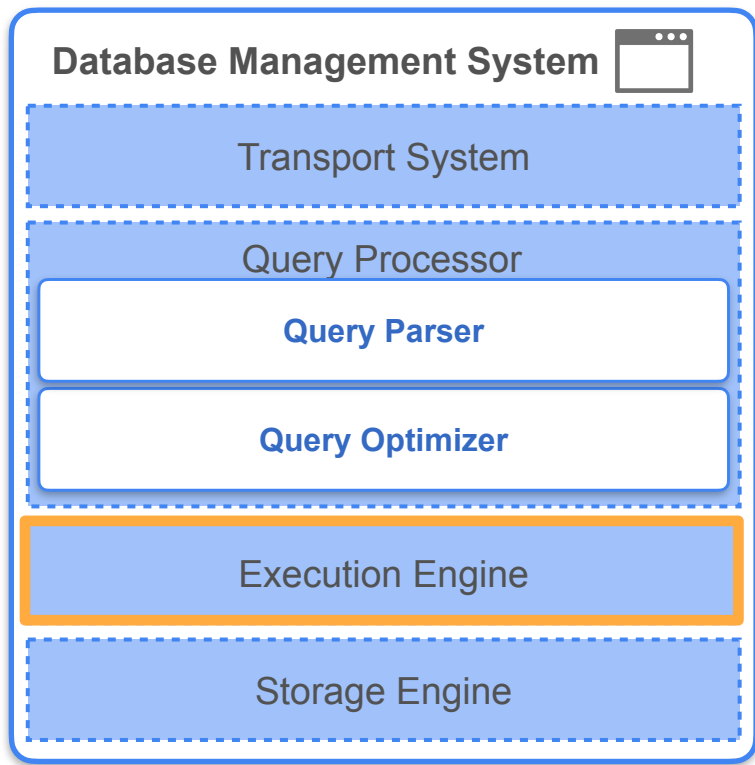
Finds a strategy that uses available resources most efficiently

0384fa1b059c

Generates various plans

- Considerations:
  - types of operations
  - presence of indexes
  - data scan size
- Calculates cost:
  - data transfer I/O cost
  - computation and memory usage cost
- Picks the least expensive execution plan

# Life of a Query



Execution plan → Query results

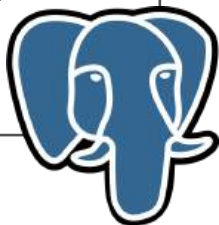
# Understanding Query Performance

## **EXPLAIN**

- Sequence of steps to execute the query
- Resource consumption
- Performance statistics in each query stage

# Understanding Query Performance

customer
customer_id
store_id
first_name
last_name
email
address_id
activebool
create_date
last_update
active



PostgreSQL

```
EXPLAIN SELECT * FROM customer;
```

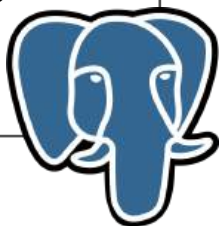
QUERY PLAN

---

Seq Scan on customer (cost=0.00..14.99 rows=599 width=70)

# Understanding Query Performance

customer
customer_id
store_id
first_name
last_name
email
address_id
activebool
create_date
last_update
active



PostgreSQL

```
EXPLAIN SELECT * FROM customer;
```

QUERY PLAN

Seq Scan on customer (cost=0.00..14.99 rows=599 width=70)


startup cost for  
query processing

total cost for  
execution



# Understanding Query Performance

customer
customer_id
store_id
first_name
last_name
email
address_id
activebool
create_date
last_update
active



PostgreSQL

```
EXPLAIN SELECT * FROM customer;
```

QUERY PLAN

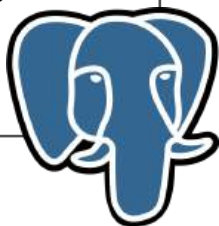
Seq Scan on customer (cost=0.00..14.99 rows=599 width=70)

estimated number  
of returned rows

expected row  
width

# Understanding Query Performance

customer
customer_id
store_id
first_name
last_name
email
address_id
activebool
create_date
last_update
active



PostgreSQL

```
EXPLAIN SELECT * FROM customer;
```

QUERY PLAN

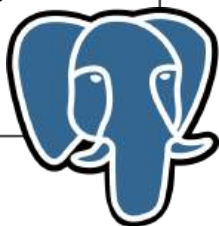
---

Seq Scan on customer (cost=0.00..14.99 rows=599 width=70)

Startup cost: 0 cost units  
Total cost: 14.99 cost units  
Return 599 rows

# Understanding Query Performance

customer
customer_id
store_id
first_name
last_name
email
address_id
activebool
create_date
last_update
active



PostgreSQL

```
EXPLAIN SELECT * FROM customer;
```

QUERY PLAN

Seq Scan on customer (cost=0.00..14.99 rows=599 width=70)

Startup cost: 0 cost units  
Total cost: 14.99 cost units  
Return 599 rows

## Index

A data structure that helps you efficiently locate data

```
EXPLAIN SELECT * FROM customer WHERE customer_id = 3;
```

QUERY PLAN

Index Scan using customer\_pkey on customer  
(cost=0.28..8.29 rows=1 width=70)

total cost



DeepLearning.AI

# Queries

---

## **Advanced SQL Queries (Part 1)**

## Data Manipulation Operations

CREATE

INSERT  
INTO

UPDATE

DELETE

## Common SQL Commands

SELECT

COUNT, SUM, AVG,  
MIN and MAX

FROM

JOIN

WHERE

GROUP BY

ORDER BY

LIMIT

## Advanced SQL Statements

SELECT  
DISTINCT

SQL  
Functions

CASE

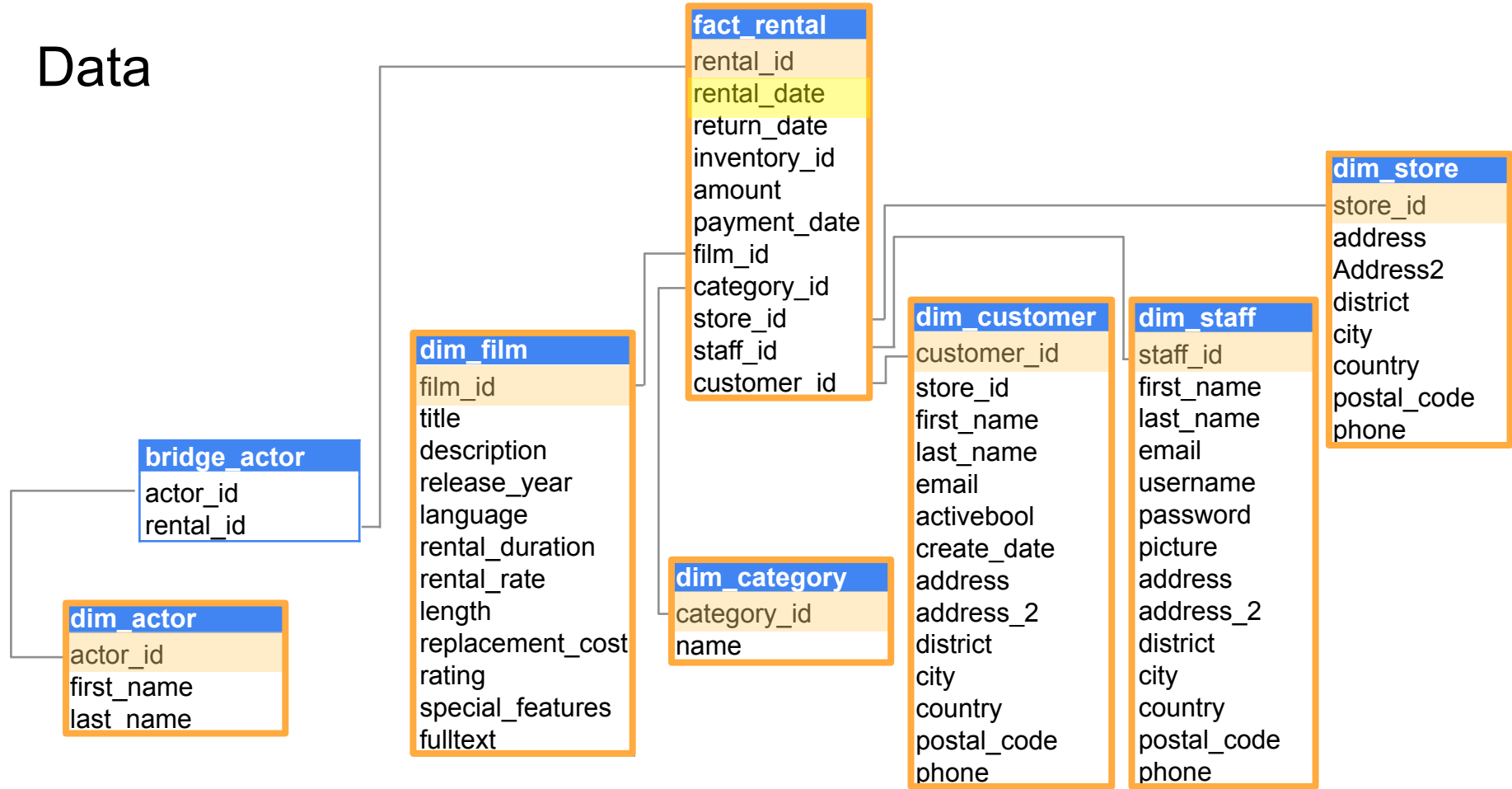
SQL Boolean  
Expressions

Common Table  
Expressions (CTE)

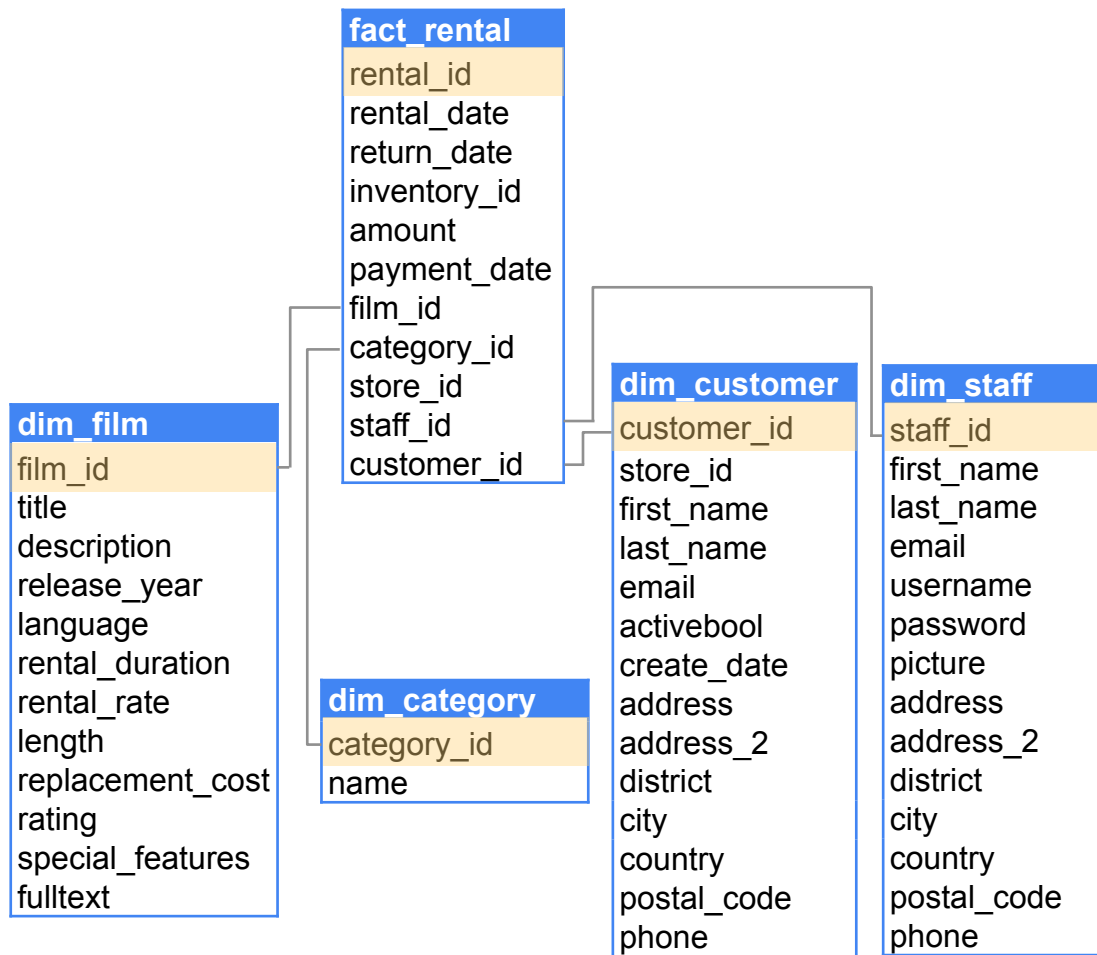
Subqueries

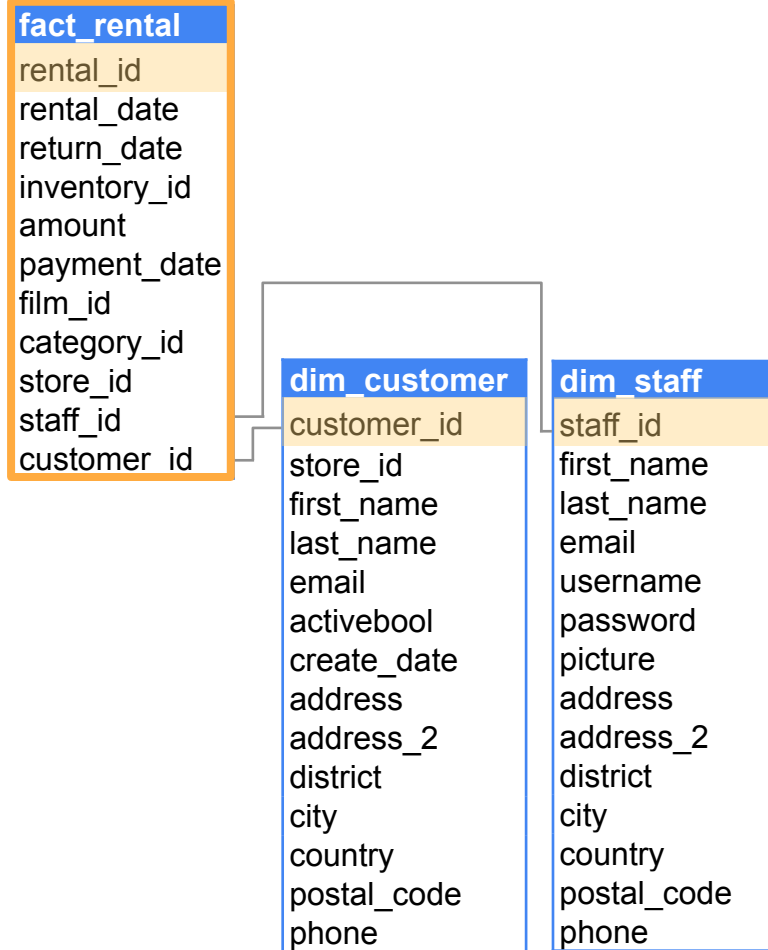
SQL Window  
Functions

# Data



# Data





Get which staff member served which customer.

```
In [ ]: %%sql
        SELECT |
        FROM fact_rental
```

```
In [ ]: 
```



**fact\_rental**

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

**dim\_customer**

customer\_id

store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

**dim\_staff**

staff\_id

first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

**SELECT  
DISTINCT**

Get which staff member served which customer.

```
In [2]: %%sql
        SELECT |
        fact_rental.staff_id,
        fact_rental.customer_id
        FROM fact_rental
```

```
* mysql+pymysql://root:***@localhost:3306/sakila_star
16044 rows affected.
```

```
Out[2]:
```

staff_id	customer_id
1	130
1	459
1	408
2	333
1	222
1	549

**fact\_rental**

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

**dim\_customer**

customer\_id

store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

**dim\_staff**

staff\_id

first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

**SELECT  
DISTINCT**

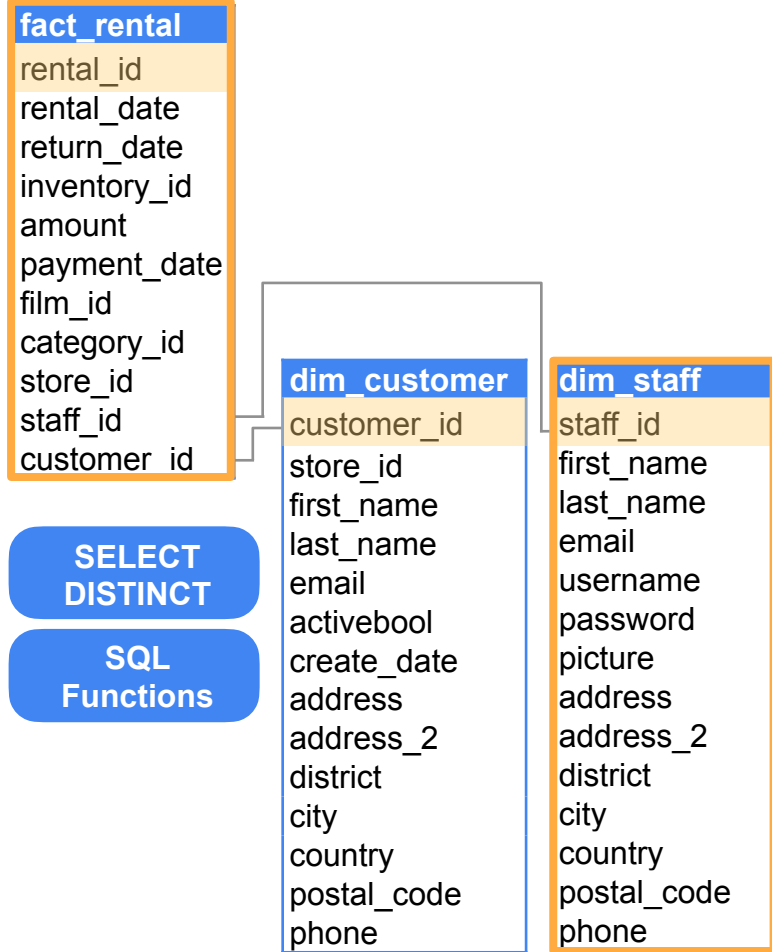
Get which staff member served which customer.

```
In [3]: %%sql
        SELECT DISTINCT
        fact_rental.staff_id,
        fact_rental.customer_id
        FROM fact_rental

        * mysql+pymysql://root:***@localhost:3306/sakila_star
        1198 rows affected.
```

```
Out[3]:
```

staff_id	customer_id
1	130
1	459
1	408
2	333
1	222
1	549



Get which staff member served which customer.

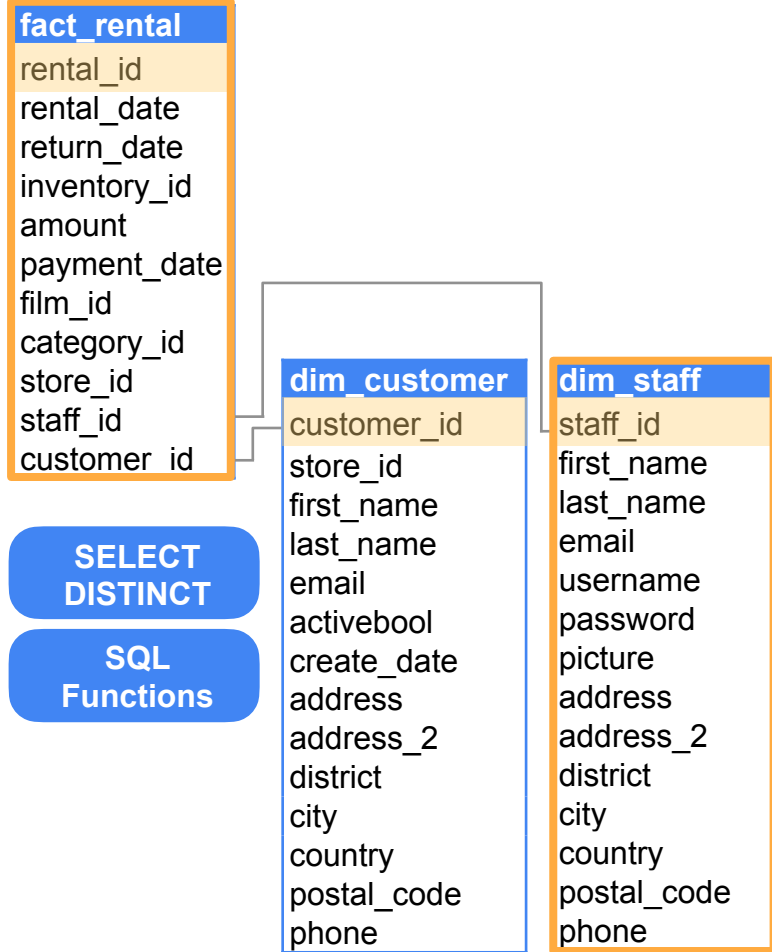
```
In [2]: %%sql
SELECT DISTINCT
fact_rental.staff_id,
dim_staff.first_name, dim_staff.last_name,
fact_rental.customer_id
FROM fact_rental
JOIN dim_staff ON fact_rental.staff_id = dim_staff.staff_id
```

```
* mysql+pymysql://root:***@localhost:3306/sakila_star
1198 rows affected.
```

```
Out[2]:
```

staff_id	customer_id
1	130
1	459
1	408
2	333
1	222

CONCAT()    LOWER()    UPPER()



Get which staff member served which customer.

```
In [3]: %%sql
SELECT DISTINCT
fact_rental.staff_id,
CONCAT(dim_staff.first_name, ' ', dim_staff.last_name) AS staff_name,
fact_rental.customer_id
FROM fact_rental
JOIN dim_staff ON fact_rental.staff_id = dim_staff.staff_id
```

```
* mysql+pymysql://root:***@localhost:3306/sakila_star
1198 rows affected.
```

```
Out[3]:
```

staff_id	staff_name	customer_id
1	Mike Hillyer	130
1	Mike Hillyer	459
1	Mike Hillyer	408
1	Mike Hillyer	222
1	Mike Hillyer	549

CONCAT()    LOWER()    UPPER()    SUBSTR()

fact_rental
rental_id
rental_date
return_date
inventory_id
amount
payment_date
film_id
category_id
store_id
staff_id
customer_id

```

CASE
WHEN cond1 THEN result1
WHEN cond2 THEN result2
ELSE result3
END

```

Check whether a customer made an on-time payment.

SELECT  
DISTINCT

SQL  
Functions

CASE

dim_customer
customer_id
store_id
first_name
last_name
email
activebool
create_date
address
address_2
district
city
country
postal_code
phone

dim_staff
staff_id
first_name
last_name
email
username
password
picture
address
address_2
district
city
country
postal_code
phone

fact_rental
rental_id
rental_date
return_date
inventory_id
amount
payment_date
film_id
category_id
store_id
staff_id
customer_id

```

CASE
WHEN cond1 THEN result1
WHEN cond2 THEN result2
ELSE result3
END

```

dim_customer
customer_id
store_id
first_name
last_name
email
activebool
create_date
address
address_2
district
city
country
postal_code
phone

dim_staff
staff_id
first_name
last_name
email
username
password
picture
address
address_2
district
city
country
postal_code
phone

SELECT  
DISTINCT

SQL  
Functions

CASE

Check whether a customer made an on-time payment.

1	Mike H	446
1	Mike H	319
1	Mike H	575

```

In [ ]: %%sql
        SELECT
        |
        FROM fact_rental

```

In [ ]:

fact_rental
rental_id
rental_date
return_date
inventory_id
amount
payment_date
film_id
category_id
store_id
staff_id
customer_id

```

CASE
WHEN cond1 THEN result1
WHEN cond2 THEN result2
ELSE result3
END

```

dim_customer
customer_id
store_id
first_name
last_name
email
activebool
create_date
address
address_2
district
city
country
postal_code
phone

dim_staff
staff_id
first_name
last_name
email
username
password
picture
address
address_2
district
city
country
postal_code
phone

SELECT  
DISTINCT

SQL  
Functions

CASE

SQL Boolean  
Expressions

## Check whether a customer made an on-time payment.

- Customers located in the United States and Canada
- Rentals occurred between Mayo 24, 2005 and July 26, 2005

```

In [5]: %%sql
        SELECT
        fact_rental.customer_id,
        fact_rental.rental_id,
        (CASE
        WHEN payment_date < return_date THEN 1
        ELSE 0
        END) AS on_time_payment
        FROM fact_rental
        LIMIT 5

* mysql+pymysql://root:***@localhost:3306/sakila_star
5 rows affected.

```

```

Out[5]:
customer_id  rental_id  on_time_payment
130          1          0
459          2          0
408          3          0
333          4          0

```

fact_rental
rental_id
rental_date
return_date
inventory_id
amount
payment_date
film_id
category_id
store_id
staff_id
customer_id

SELECT  
DISTINCT

SQL  
Functions

CASE

SQL Boolean  
Expressions

CASE

```

WHEN cond1 THEN result1
WHEN cond2 THEN result2
ELSE result3
END

```

dim_customer
customer_id
store_id
first_name
last_name
email
activebool
create_date
address
address_2
district
city
country
postal_code
phone

dim_staff
staff_id
first_name
last_name
email
username
password
picture
address
address_2
district
city
country
postal_code
phone

Check whether a customer made an on-time payment.

- Customers located in the United States and Canada
- Rentals occurred between Mayo 24, 2005 and July 26, 2005

```

In [5]: %%sql
SELECT
fact_rental.customer_id,
fact_rental.rental_id,
(CASE
WHEN payment_date<return_date THEN 1
ELSE 0
END) AS on_time_payment
FROM fact_rental
JOIN dim_customer ON dim_customer.customer_id=fact_rental.customer_id
LIMIT 5

```

```

* mysql+pymysql://root:***@localhost:3306/sakila_star
5 rows affected.

```

Out[5]:

customer_id	rental_id	on_time_payment
130	1	0
459	2	0
408	3	0



fact_rental
rental_id
rental_date
return_date
inventory_id
amount
payment_date
film_id
category_id
store_id
staff_id
customer_id

```

CASE
WHEN cond1 THEN result1
WHEN cond2 THEN result2
ELSE result3
END

```

dim_customer
customer_id
store_id
first_name
last_name
email
activebool
create_date
address
address_2
district
city
country
postal_code
phone

dim_staff
staff_id
first_name
last_name
email
username
password
picture
address
address_2
district
city
country
postal_code
phone

SELECT  
DISTINCT

SQL  
Functions

CASE

SQL Boolean  
Expressions

## Check whether a customer made an on-time payment.

- Customers located in the United States and Canada
- Rentals occurred between Mayo 24, 2005 and July 26, 2005

```

In [5]: %%sql
        SELECT
        fact_rental.customer_id,
        fact_rental.rental_id,
        (CASE
        WHEN payment_date < return_date THEN 1
        ELSE 0
        END) AS on_time_payment
        FROM fact_rental
        JOIN dim_customer ON dim_customer.customer_id=fact_rental.customer_id
        WHERE dim_customer.country="United States"
        OR dim_customer.country= "Canada"
        LIMIT 5

```

```

* mysql+pymysql://root:***@localhost:3306/sakila_star
5 rows affected.

```

```

Out[5]:
customer_id  rental_id  on_time_payment
-----
130          1          0
459          2          0

```

fact_rental
rental_id
rental_date
return_date
inventory_id
amount
payment_date
film_id
category_id
store_id
staff_id
customer_id

```

CASE
WHEN cond1 THEN result1
WHEN cond2 THEN result2
ELSE result3
END

```

dim_customer
customer_id
store_id
first_name
last_name
email
activebool
create_date
address
address_2
district
city
country
postal_code
phone

dim_staff
staff_id
first_name
last_name
email
username
password
picture
address
address_2
district
city
country
postal_code
phone

SELECT  
DISTINCT

SQL  
Functions

CASE

SQL Boolean  
Expressions

## Check whether a customer made an on-time payment.

- Customers located in the United States and Canada
- Rentals occurred between Mayo 24, 2005 and July 26, 2005

```

In [5]: %%sql
SELECT
fact_rental.customer_id,
fact_rental.rental_id,
(CASE
WHEN payment_date < return_date THEN 1
ELSE 0
END) AS on_time_payment
FROM fact_rental
JOIN dim_customer ON dim_customer.customer_id=fact_rental.customer_id
WHERE dim_customer.country IN ("United States", "Canada")
LIMIT 5

* mysql+pymysql://root:***@localhost:3306/sakila_star
5 rows affected.

```

```

Out[5]:
customer_id  rental_id  on_time_payment
-----
130          1          0
459          2          0

```

# Advanced SQL Statements

**SELECT  
DISTINCT**

**SQL  
Functions**

**CASE**

**SQL Boolean  
Expressions**



DeepLearning.AI

# Queries

---

## **Advanced SQL Queries (Part 2)**

# Advanced SQL Statements

```
SELECT DISTINCT
fact_rental.staff_id,
CONCAT (dim_staff.first_name, ' ',
         dim_staff.last_name) AS staff_name,
fact_rental.customer_id
FROM fact_rental
JOIN dim_staff
ON fact_rental.staff_id = dim_staff.staff_id
```

```
SELECT fact_rental.customer_id, fact_rental.rental_id,
(CASE WHEN payment_date < return_date THEN 1
      ELSE 0
END) AS on_time_payment
FROM fact_rental
JOIN dim_customer
ON dim_customer.customer_id = fact_rental.customer_id
WHERE dim_customer.country IN ("United States", "Canada")
AND
(fact_rental.rental_date between "2005-05-24" and "2005-07-26")
```

Find the total number of customers served by each staff member

```
SELECT DISTINCT  
fact_rental.staff_id,  
CONCAT (dim_staff.first_name, ' ',  
         dim_staff.last_name) AS staff_name,  
fact_rental.customer_id  
FROM fact_rental  
JOIN dim_staff  
ON fact_rental.staff_id = dim_staff.staff_id
```

staff_id	staff_name	customer_id
1	Mike Hillyer	130
1	Mike Hillyer	459
1	Mike Hillyer	408
1	Mike Hillyer	222
1	Mike Hillyer	549

Use Common Table Expressions (CTE)  
to define these temporary results

```
SELECT fact_rental.customer_id, fact_rental.rental_id,  
(CASE WHEN payment_date < return_date THEN 1  
  ELSE 0  
  END) AS on_time_payment  
FROM fact_rental  
JOIN dim_customer  
ON dim_customer.customer_id = fact_rental.customer_id  
WHERE dim_customer.country IN ("United States", "Canada")  
AND  
(fact_rental.rental_date between "2005-05-24" and "2005-07-26")
```

customer_id	rental_id	on_time_payment
2	320	0
2	2128	0
2	5636	0
2	5755	0
6	57	0

Compute for each customer the  
average of on\_time\_payment

### fact\_rental

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

### Common Table Expressions

### dim\_customer

customer\_id  
store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

### dim\_staff

staff\_id  
first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

Find the total number of customers served by each staff member

```
In [ ]: %%sql
```

### fact\_rental

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

### dim\_customer

customer\_id  
store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

### dim\_staff

staff\_id  
first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

### Common Table Expressions

Find the total number of customers served by each staff member

```
In [ ]: %%sql
        WITH staff_customer_pairs AS (
          SELECT DISTINCT fact_rental.staff_id,
            CONCAT(dim_staff.first_name, ' ', dim_staff.last_name) AS staff_name,
            fact_rental.customer_id
          FROM fact_rental
          JOIN dim_staff ON fact_rental.staff_id = dim_staff.staff_id
        )
```



## fact\_rental

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

## Common Table Expressions

## dim\_customer

customer\_id  
store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## dim\_staff

staff\_id  
first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

Compute the percentage of on-time payments for each customer.

In [ ]: `%%sql`

## fact\_rental

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

## dim\_customer

customer\_id  
store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## dim\_staff

staff\_id  
first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## Common Table Expressions

Compute the percentage of on-time payments for each customer.

```
In [ ]: %%sql
WITH customer_payment_info AS (
  SELECT fact_rental.customer_id, fact_rental.rental_id,
  (CASE
    WHEN payment_date < return_date THEN 1
    ELSE 0
  END) AS on_time_payment
  FROM fact_rental
  JOIN dim_customer ON dim_customer.customer_id=fact_rental.customer_id
  WHERE dim_customer.country IN ("United States", "Canada")
  AND (fact_rental.rental_date between "2005-05-24" and "2005-07-26")
)
```

## fact\_rental

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

## dim\_customer

customer\_id  
store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## dim\_staff

staff\_id  
first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## Common Table Expressions

Compute the percentage of on-time payments for each customer.

```
In [ ]: %%sql
WITH customer_payment_info AS (
  SELECT fact_rental.customer_id, fact_rental.rental_id,
  (CASE
    WHEN payment_date < return_date THEN 1
    ELSE 0
  END) AS on_time_payment
  FROM fact_rental
  JOIN dim_customer ON dim_customer.customer_id=fact_rental.customer_id
  WHERE dim_customer.country IN ("United States", "Canada")
  AND (fact_rental.rental_date between "2005-05-24" and "2005-07-26")
)
SELECT customer_id, AVG(on_time_payment) AS percent_on_time_payment
FROM customer_payment_info
```

## fact\_rental

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

## dim\_customer

customer\_id  
store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## dim\_staff

staff\_id  
first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## Common Table Expressions

Find the maximum of the “percent on time payment” column.

```
WITH customer_payment_info AS (  
  SELECT fact_rental.customer_id, fact_rental.rental_id,  
  (CASE  
    WHEN payment_date < return_date THEN 1  
    ELSE 0  
  END) AS on_time_payment  
  FROM fact_rental  
  JOIN dim_customer ON dim_customer.customer_id=fact_rental.customer_id  
  WHERE dim_customer.country IN ("United States", "Canada")  
  AND (fact_rental.rental_date between "2005-05-24" and "2005-07-26")  
)  
SELECT customer_id, AVG(on_time_payment) AS percent_on_time_payment  
FROM customer_payment_info  
GROUP BY customer_id
```

```
* mysql+pymysql://root:***@localhost:3306/sakila_star  
41 rows affected.
```

```
Out[3]:
```

customer_id	percent_on_time_payment
2	0.0000

## fact\_rental

rental\_id  
rental\_date  
return\_date  
inventory\_id  
amount  
payment\_date  
film\_id  
category\_id  
store\_id  
staff\_id  
customer\_id

## dim\_customer

customer\_id  
store\_id  
first\_name  
last\_name  
email  
activebool  
create\_date  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## dim\_staff

staff\_id  
first\_name  
last\_name  
email  
username  
password  
picture  
address  
address\_2  
district  
city  
country  
postal\_code  
phone

## Common Table Expressions

Find the maximum of the “percent on time payment” column.

```
WITH customer_payment_info AS (  
  SELECT fact_rental.customer_id, fact_rental.rental_id,  
  (CASE  
    WHEN payment_date < return_date THEN 1  
    ELSE 0  
  END) AS on_time_payment  
  FROM fact_rental  
  JOIN dim_customer ON dim_customer.customer_id=fact_rental.customer_id  
  WHERE dim_customer.country IN ("United States", "Canada")  
  AND (fact_rental.rental_date between "2005-05-24" and "2005-07-26")  
) , customer_percent_on_time_payment AS (  
  SELECT customer_id, AVG(on_time_payment) AS percent_on_time_payment  
  FROM customer_payment_info  
  GROUP BY customer_id  
)  
|
```

```
mysql+pymysql://root:***@localhost:3306/sakila_star  
41 rows affected.
```

```
Out[5]:  customer_id  percent_on_time_payment
```

## Subqueries

dim_film
film_id
title
description
release_year
language
rental_duration
rental_rate
length
replacement_cost
rating
special_features
fulltext

fact_rental
rental_id
rental_date
return_date
inventory_id
amount
payment_date
film_id
category_id
store_id
staff_id
customer_id

dim_category
category_id
name

Get the ids of the films that have length greater than the average length.

## Subqueries

**dim\_film**

- film\_id
- title
- description
- release\_year
- language
- rental\_duration
- rental\_rate
- length
- replacement\_cost
- rating
- special\_features
- fulltext

**fact\_rental**

- rental\_id
- rental\_date
- return\_date
- inventory\_id
- amount
- payment\_date
- film\_id
- category\_id
- store\_id
- staff\_id
- customer\_id

**dim\_category**

- category\_id
- name

Get the ids of the films that have length greater than the average length.

```
In [8]: %%sql
        SELECT AVG(length) from dim_film

        * mysql+pymysql://root:***@localhost:3306/sakila_star
        1 rows affected.
```


```
Out[8]:
        AVG(length)
        115.2720
```

```
In [ ]: %%sql
        |
```

## SQL Window Functions

- Allows you to apply an aggregate or ranking function over a particular window or a set of rows.
- Does not group rows into a single output row: each row remains separate

rank()  
row\_number()



```
SELECT column_name1,  
ranking_function() OVER (  
    PARTITION BY column_name1  
    ORDER BY column_name3)  
AS new_column  
FROM table_name;
```



**fact\_rental**

rental\_id

rental\_date

return\_date

inventory\_id

amount

payment\_date

film\_id

category\_id

store\_id

staff\_id

customer\_id

## SQL Window Functions

```
SELECT column_name1,  
       ranking_function() OVER (  
    PARTITION BY column_name  
    ORDER BY column_name3)  
AS new_column  
FROM table_name
```

**dim\_customer**

customer\_id

store\_id

first\_name

last\_name

email

activebool

create\_date

address

address\_2

district

city

country

postal\_code

phone

**dim\_category**

category\_id

name

Compute the average duration in days that a customer spent on a film category.

```
In [ ]: %%sql  
SELECT  
  
FROM fact_rental  
JOIN dim_category  
ON fact_rental.category_id = dim_category.category_id
```

**fact\_rental**

rental\_id

rental\_date

return\_date

inventory\_id

amount

payment\_date

film\_id

category\_id

store\_id

staff\_id

customer\_id

## SQL Window Functions

```
SELECT column_name1,  
       ranking_function() OVER (  
    PARTITION BY column_name  
    ORDER BY column_name3)  
AS new_column  
FROM table_name
```

**dim\_customer**

customer\_id

store\_id

first\_name

last\_name

email

activebool

create\_date

address

address\_2

district

city

country

postal\_code

phone

**dim\_category**

category\_id

name

Compute the average duration in days that a customer spent on a film category.

```
In [ ]: %%sql  
SELECT  
    fact_rental.customer_id,  
    dim_category.name,  
    avg(datediff(return_date, rental_date)) AS average_rental_days  
FROM fact_rental  
JOIN dim_category  
ON fact_rental.category_id = dim_category.category_id  
|
```

fact\_rental

rental\_id

rental\_date

return\_date

inventory\_id

amount

payment\_date

film\_id

category\_id

store\_id

staff\_id

customer\_id

## SQL Window Functions

```
SELECT column_name1,  
ranking_function() OVER (  
PARTITION BY column_name  
ORDER BY column_name3)  
AS new_column  
FROM table_name
```

dim\_customer

customer\_id

store\_id

first\_name

last\_name

email

activebool

create\_date

address

address\_2

district

city

country

postal\_code

phone

dim\_category

category\_id

name

Add a column that shows for each customer the rank of each category based on the rental days.

```
n [10]:  %%sql  
SELECT  
fact_rental.customer_id,  
dim_category.name,  
avg(datediff(return_date, rental_date)) AS average_rental_days  
FROM fact_rental  
JOIN dim_category  
ON fact_rental.category_id = dim_category.category_id  
GROUP BY fact_rental.customer_id, dim_category.name  
ORDER BY fact_rental.customer_id, average_rental_days DESC
```

Out[10]:

customer_id	name	average_rental_days
1	Documentary	9.0000
1	Travel	8.0000
1	Games	6.0000
1	Music	5.5000
1	Comedy	5.4000

fact\_rental

rental\_id

rental\_date

return\_date

inventory\_id

amount

payment\_date

film\_id

category\_id

store\_id

staff\_id

customer\_id

## SQL Window Functions

```
SELECT column_name1,  
ranking_function() OVER (  
PARTITION BY column_name  
ORDER BY column_name3)  
AS new_column  
FROM table_name
```

dim\_customer

customer\_id

store\_id

first\_name

last\_name

email

activebool

create\_date

address

address\_2

district

city

country

postal\_code

phone

dim\_category

category\_id

name

Add a column that shows for each customer the rank of each category based on the rental days.

```
n [10]:  %%sql  
WITH customer_info AS (  
SELECT  
fact_rental.customer_id,  
dim_category.name,  
avg(datediff(return_date, rental_date)) AS average_rental_days  
FROM fact_rental  
JOIN dim_category  
ON fact_rental.category_id = dim_category.category_id  
GROUP BY fact_rental.customer_id, dim_category.name  
ORDER BY fact_rental.customer_id, average_rental_days DESC  
)  
|
```

```
* mysql+pymysql://root:***@localhost:3306/sakila_star  
7741 rows affected.
```

```
Out[10]:
```

customer_id	name	average_rental_days
1	Documentary	9.0000
4	Travel	8.0000

fact\_rental

rental\_id

rental\_date

return\_date

inventory\_id

amount

payment\_date

film\_id

category\_id

store\_id

staff\_id

customer\_id

## SQL Window Functions

```
SELECT column_name1,  
ranking_function() OVER (  
PARTITION BY column_name  
ORDER BY column_name3)  
AS new_column  
FROM table_name
```

dim\_customer

customer\_id

store\_id

first\_name

last\_name

email

activebool

create\_date

address

address\_2

district

city

country

postal\_code

phone

dim\_category

category\_id

name

Add a column that shows for each customer the rank of each category based on the rental days.

```
In [10]: %%sql  
WITH customer_info AS (  
SELECT  
fact_rental.customer_id,  
dim_category.name,  
avg(datediff(return_date, rental_date)) AS average_rental_days  
FROM fact_rental  
JOIN dim_category  
ON fact_rental.category_id = dim_category.category_id  
GROUP BY fact_rental.customer_id, dim_category.name  
ORDER BY fact_rental.customer_id, average_rental_days DESC  
)  
SELECT customer_id, name, average_rental_days  
FROM customer_info
```

```
* mysql+pymysql://root:***@localhost:3306/sakila_star  
7741 rows affected.
```

```
Out[10]: customer_id      name  average_rental_days
```

fact\_rental

rental\_id

rental\_date

return\_date

inventory\_id

amount

payment\_date

film\_id

category\_id

store\_id

staff\_id

customer\_id

## SQL Window Functions

```
SELECT column_name1,  
ranking_function() OVER (  
PARTITION BY column_name  
ORDER BY column_name3)  
AS new_column  
FROM table_name
```

dim\_customer

customer\_id

store\_id

first\_name

last\_name

email

activebool

create\_date

address

address\_2

district

city

country

postal\_code

phone

dim\_category

category\_id

name

Add a column that shows for each customer the rank of each category based on the rental days.

```
n [10]:  %%sql  
WITH customer_info AS (  
SELECT  
fact_rental.customer_id,  
dim_category.name,  
avg(datediff(return_date, rental_date)) AS average_rental_days  
FROM fact_rental  
JOIN dim_category  
ON fact_rental.category_id = dim_category.category_id  
GROUP BY fact_rental.customer_id, dim_category.name  
ORDER BY fact_rental.customer_id, average_rental_days DESC  
)  
SELECT customer_id, name, average_rental_days,  
rank() OVER  
(PARTITION BY customer_id ORDER BY average_rental_days DESC)  
AS rank_category  
FROM customer_info  
  
* mysql+pymysql://root:***@localhost:3306/sakila_star  
7741 rows affected.
```

rank(): assigns same rank to rows there's a tie  
row\_number(): assigns different ranks when there's a tie

fact\_rental

rental\_id

rental\_date

return\_date

inventory\_id

amount

payment\_date

film\_id

category\_id

store\_id

staff\_id

customer\_id

## SQL Window Functions

```
SELECT column_name1,  
ranking_function() OVER (  
PARTITION BY column_name  
ORDER BY column_name3)  
AS new_column  
FROM table_name
```

dim\_customer

customer\_id

store\_id

first\_name

last\_name

email

activebool

create\_date

address

address\_2

district

city

country

postal\_code

phone

dim\_category

category\_id

name

Add a column that shows for each customer the running sum over each window.

```
n [11]: %%sql  
WITH customer_info AS (  
SELECT  
fact_rental.customer_id,  
dim_category.name,  
avg(datediff(return_date, rental_date)) AS average_rental_days  
FROM fact_rental  
JOIN dim_category  
ON fact_rental.category_id = dim_category.category_id  
GROUP BY fact_rental.customer_id, dim_category.name  
ORDER BY fact_rental.customer_id, average_rental_days DESC  
)  
SELECT customer_id, name, average_rental_days,  
rank() OVER  
(PARTITION BY customer_id ORDER BY average_rental_days DESC)  
AS rank_category  
FROM customer_info  
ORDER BY customer_id, rank_category  
  
* mysql+pymysql://root:***@localhost:3306/sakila_star  
7741 rows affected
```



DeepLearning.AI

# Queries

---

## Index Deep Dive



# Index

## Index

A separate data structure that has its own disk space and contains information that refers to the actual table

**Book index:** quickly find pages instead of flipping through the entire book

**DBMS:** queries data using an ordered index rather than scanning the whole table

## Index

### Symbols

1NF (first normal form), 291

2NF (second normal form), 291

3NF (third normal form), 291

### A

abstraction, 22

access policies, 376

accountability, 55

accuracy, 55

ACID (atomicity, consistency, isolation, and durability) transactions, 103, 158

active security, 371, 377

address analysis, 345

application architecture, 72 (see also data architecture; monolithic architectures; technical architectures)

application databases, 157-159

application programming interfaces (APIs), 157, 174-176, 254

architecture tiers, 90

archival storage, 197

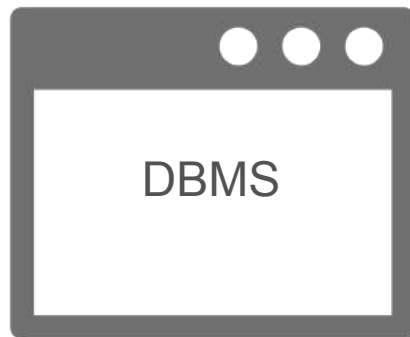
areal density, 192

asymmetric optimization, 148

asynchronous data ingestion, 238

atomic transactions, 157-158

atomicity, consistency, isolation, and durability (ACID) transactions, 103, 158



# Index

```
SELECT * FROM order WHERE country = 'USA'
```

Order ID	Price	Product SKU	Quantity	Customer ID	Store ID	Country
2	23	902348	14	56t	3	Mexico
3	56	458645	13	69t	3	Chile
4	23	902348	14	56t	3	Canada
5	45	1255893	12	87q	4	Canada
6	50	456829	13	98q	1	USA
7	34	568298	12	98q	1	USA
8	44	563783	4	67t	1	Canada
9	22	234589	5	56u	2	Brazil
10	30	267895	12	78y	3	USA
11	60	545659	14	13t	5	Mexico
....	....	....	....	....	....	....

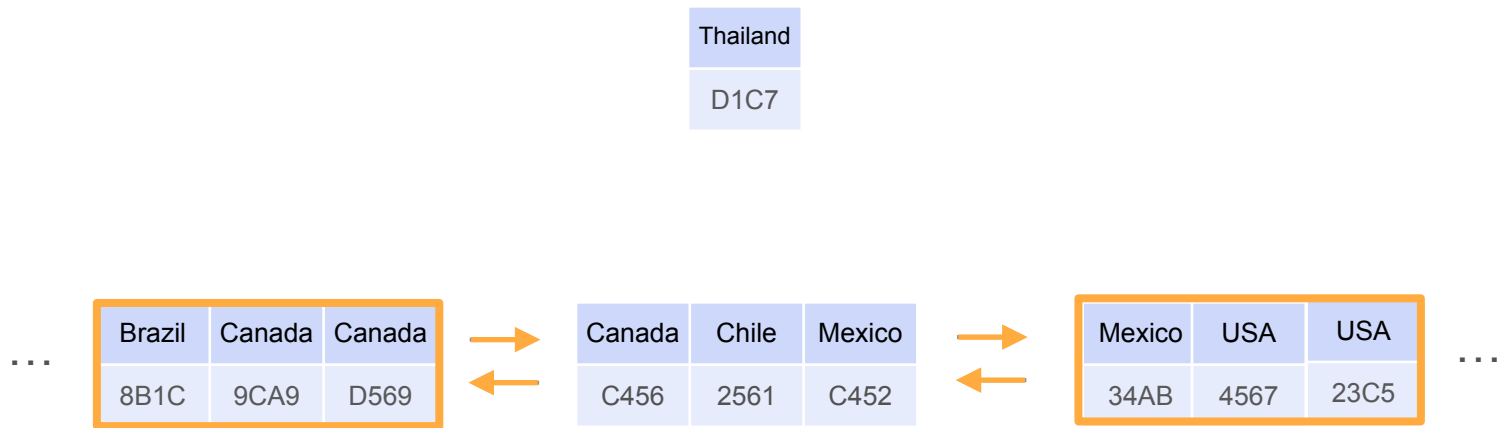
Use binary search to locate the USA rows

## Index

Country	Row Address
...	...
Brazil	8B1C
Canada	9CA9
Canada	D569
Canada	C456
Chile	2561
Mexico	C452
Mexico	34AB
USA	4567
USA	23C5
USA	7C6E
....	....

# Index

- The physical location of the blocks does not matter
- This structure facilitates the update of the index when data is inserted or deleted

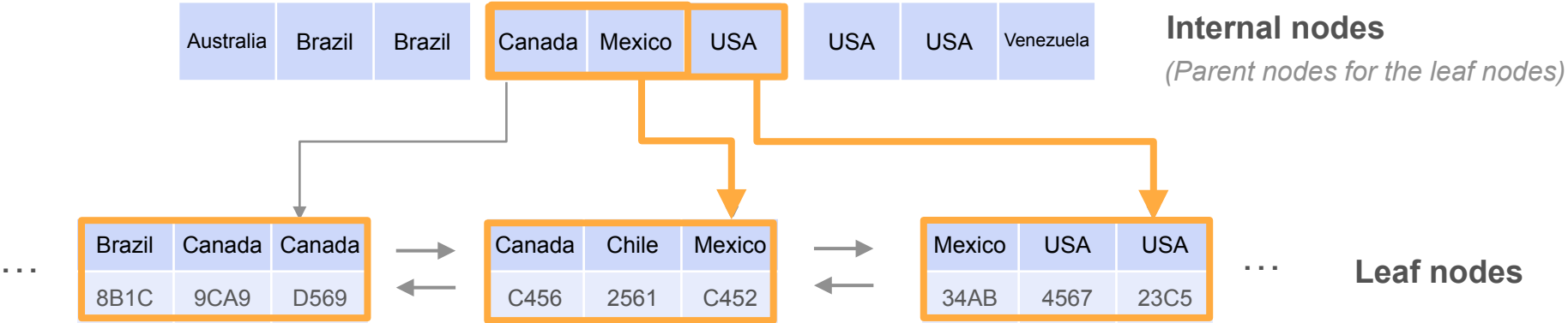


## Index

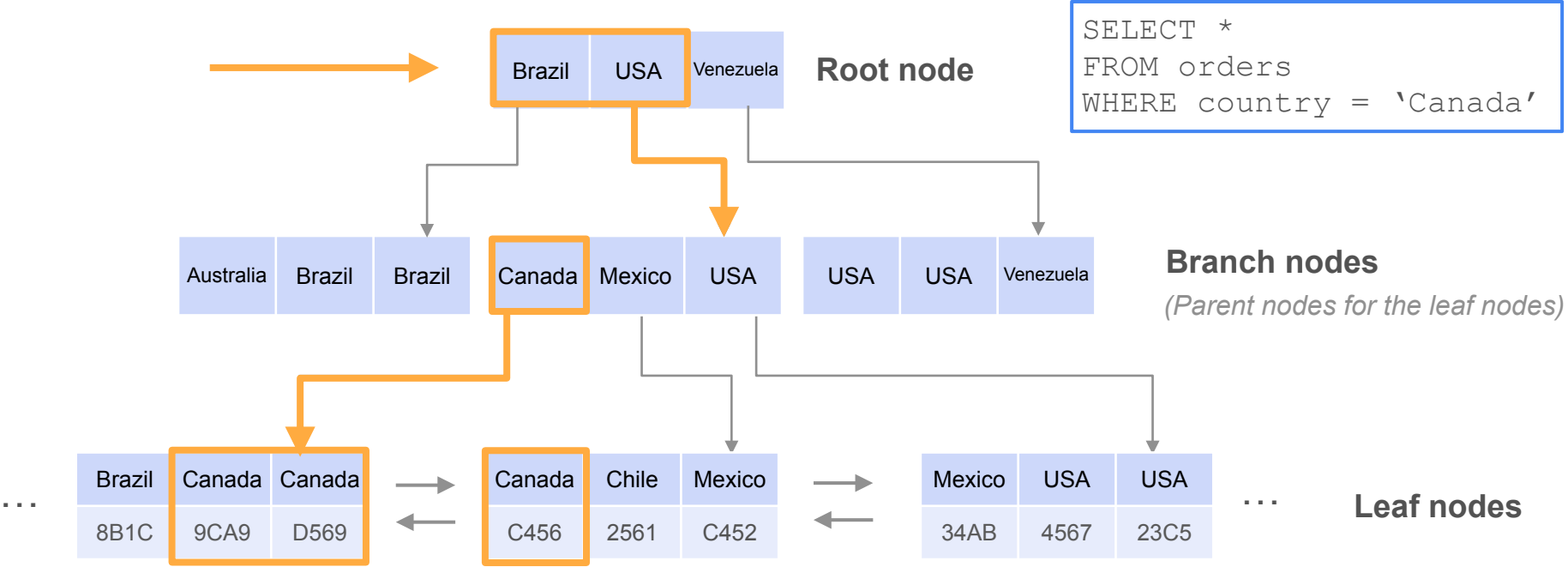
Country	Row Address
...	...
Brazil	8B1C
Canada	9CA9
Canada	D569
Canada	C456
Chile	2561
Mexico	C452
Mexico	34AB
USA	4567
USA	23C5
USA	7C6E
....	....

# Balanced Search Tree (B-Tree)

Name between ~~Mexico~~ and ~~USA~~

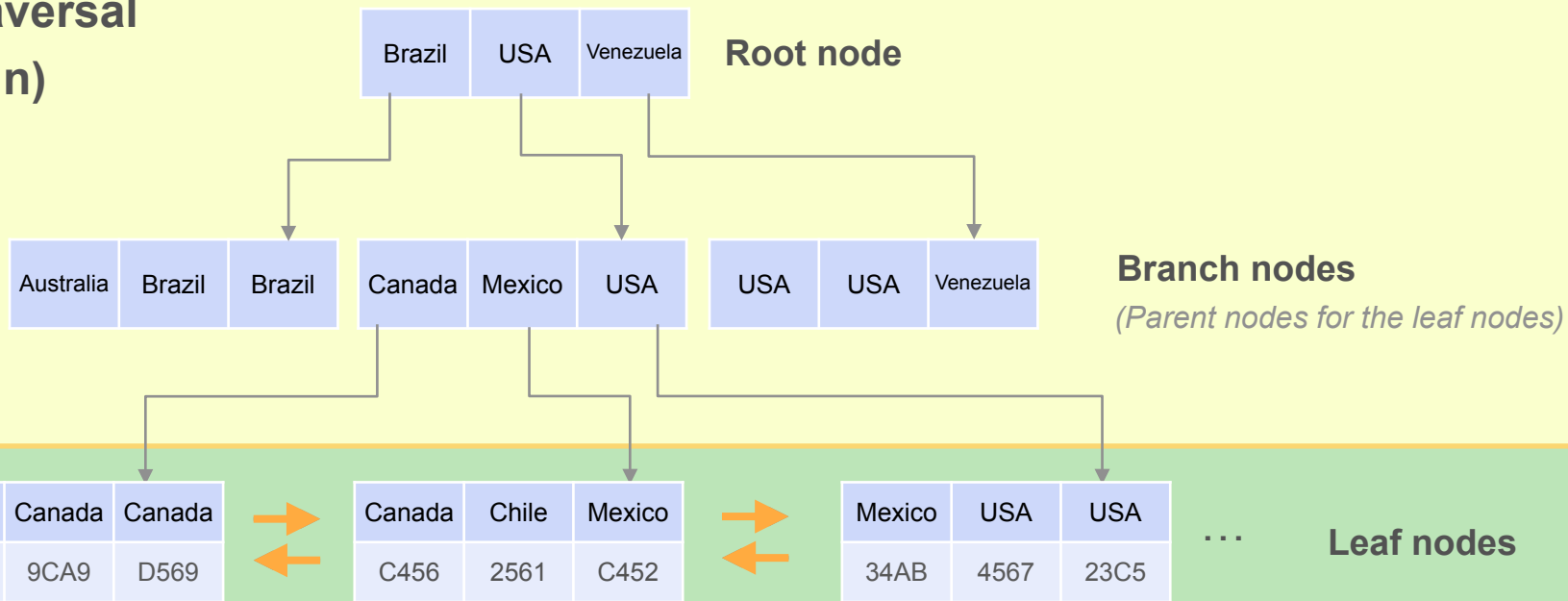


# Balanced Search Tree (B-Tree)



# Balanced Search Tree (B-Tree)

## Tree Traversal $O(\log n)$



**Chain Traversal** If there are lots of repeated values, it becomes expensive

# Index

## payment

payment\_id (*primary key*)

customer\_id

staff\_id

rental\_id

amount

payment\_date

```
EXPLAIN SELECT * FROM payment WHERE rental_id = 1;
```

QUERY PLAN

---

Seq Scan on payment (cost=0.00..290.45 rows=1 width=26)

## Create an index for the rental\_id column

```
CREATE INDEX rental_idx ON payment (rental_id);
```

```
EXPLAIN SELECT * FROM payment WHERE rental_id = 1;
```

QUERY PLAN

---

Index Scan using rental\_idx on payment  
(cost=0.29..8.30 rows=1 width=26)

# Columnar Storage



Amazon Redshift

**Sort Key:** one or more columns

- Sorts the data according to the sort key
- Stores the sorted data on disk

Sort key



Order ID	Product SKU	Quantity	Customer ID	Country
2	902348	14	56t	Mexico
3	458645	13	69t	Chile
4	902348	14	56t	Canada
5	1255893	12	87q	Canada
6	456829	13	98q	USA
7	568298	12	98q	USA
8	563783	4	67t	USA
9	234589	5	56u	Brazil
10	267895	12	78y	Canada
11	545659	14	13t	Mexico



# Columnar Storage



Amazon Redshift

**Sort Key:** one or more columns

- Sorts the data according to the sort key
- Stores the sorted data on disk

Also known as cluster key.

Sort key



Order ID	Product SKU	Quantity	Customer ID	Country
9	234589	5	56u	Brazil
3	458645	13	69t	Chile
10	267895	12	78y	Canada
4	902348	14	56t	Canada
5	1255893	12	87q	Canada
2	902348	14	56t	Mexico
11	545659	14	13t	Mexico
6	456829	13	98q	USA
7	568298	12	98q	USA
8	563783	4	67t	USA

Stored on disk



DeepLearning.AI

## Queries

---

**Retrieving Only the Data You  
Need**

# Avoid Select \*

**Query:** `SELECT * FROM order`




Order ID	Price	Product SKU	Quantity	Customer ID	Store ID	Country
1	40	458650	10	67t	3	Canada
2	23	902348	14	56t	3	Columbia
3	56	458645	13	69t	3	Chile
4	23	902348	14	56t	3	Canada
5	45	1255893	12	87q	4	Canada
6	50	456829	13	98q	1	USA
7	34	568298	12	98q	1	USA
8	44	563783	4	67t	1	USA
9	22	234589	5	56u	2	Brazil
10	30	267895	12	78y	3	Canada
11	60	545659	14	13t	5	Mexico

- Large amounts of data need to be transferred from disk.

# Avoid Select \*

Query: SELECT \*  from order



Order ID	Price	Product SKU	Quantity	Customer ID	Store ID	Country
1	40	458650	10	67t	3	Canada
2	23	902348	14	56t	3	Columbia
3	56	458645	13	69t	3	Chile
4	23	902348	14	56t	3	Canada
5	45	1255893	12	87q	4	Canada
6	50	456829	13	98q	1	USA
7	34	568298	12	98q	1	USA
8	44	563783	4	67t	1	USA
9	22	234589	5	56u	2	Brazil
10	30	267895	12	78y	3	Canada
11	60	545659	14	13t	5	Mexico

- Large amounts of data need to be transferred from disk.
- Select \* on you cloud pay-as-you-go databases can be expensive.
  - Charged for reading all bytes and for utilizing compute resources.



# Query Tips

## Pruning

Exclude irrelevant data from being scanned in your query.

### Row-based pruning

- Filter out rows
- Use index or sort/cluster key

```
CREATE INDEX rental_idx
ON payment (rental_id);

SELECT * FROM payment
WHERE rental_id = 1;
```

### Column-based pruning

- Specify the columns you need

```
SELECT customer_id,
        rental_id
FROM payment;
```

### Partition pruning

- Scan specific partitions
- Partitions are based on a partition key

# Partitioning

Sorted by Country, not partitioned

Order ID	Order Date	Product SKU	Quantity	Country
9	2023-04-07	234589	5	Brazil
4	2023-04-05	902348	14	Canada
5	2023-04-07	1255893	12	Canada
10	2023-04-01	267895	12	Canada
3	2023-04-03	458645	13	Chile
2	2023-04-01	902348	14	Mexico
11	2023-04-03	545659	14	Mexico
6	2023-04-01	456829	13	USA
7	2023-04-03	568298	12	USA
8	2023-04-05	563783	4	USA



Partition  
2023-04-01

Partition  
2023-04-03

Partition  
2023-04-05

Partition  
2023-04-07



Partitioned by Date  
Each partition is sorted by Country



DeepLearning.AI

# Queries

---

## The Join Statement

# Example on Join

```
SELECT * FROM orders
JOIN customers
ON customers.id = orders.customer_id
```

orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.



# Example on Join

```
SELECT * FROM orders
JOIN customers
ON customers.id = orders.customer_id
```

orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

Combined Table

order_id	customer_id	product_id	date_time	purchase_amount	id	first_name	last_name	address
----------	-------------	------------	-----------	-----------------	----	------------	-----------	---------

# Example on Join

```
SELECT * FROM orders
JOIN customers
ON customers.id = orders.customer_id
```

orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

Combined Table

order_id	customer_id	product_id	date_time	purchase_amount	id	first_name	last_name	address
1	1	1	12/08/2024	700	1	Jane	Doe	74th St
2	1	2	12/08/2024	99	1	Jane	Doe	74th St
3	1	3	12/08/2024	100	1	Jane	Doe	74th St
4	2	4	12/08/2024	899	2	Mary	Ann	19th Ave.
5	3	4	12/08/2024	899	3	John	Ken	2st Link
6	4	4	12/08/2024	899	4	Ivy	Tan	67th St.

# Example on Join

```
SELECT * FROM orders
JOIN customers
ON customers.id = orders.customer_id
```

## Inner Join

Combines data from only the rows that share a matching customer id in both tables

## Combined Table

order_id	customer_id	product_id	date_time	purchase_amount	id	first_name	last_name	address
1	1	1	12/08/2024	700	1	Jane	Doe	74th St
2	1	2	12/08/2024	99	1	Jane	Doe	74th St
3	1	3	12/08/2024	100	1	Jane	Doe	74th St
4	2	4	12/08/2024	899	2	Mary	Ann	19th Ave.
5	3	4	12/08/2024	899	3	John	Ken	2st Link
6	4	4	12/08/2024	899	4	Ivy	Tan	67th St.

# Method 1 - Nested Loop Join

## orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	1	4	12/08/2024	899

## customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

## Combined Table

order_id	customer_id	product_id	date_time	purchase_amount	id	first_name	last_name	address
1	1	1	12/08/2024	700	1	Jane	Doe	74th St
2	1	2	12/08/2024	99	1	Jane	Doe	74th St
3	1	3	12/08/2024	100	1	Jane	Doe	74th St
4	2	4	12/08/2024	899	2	Mary	Ann	19th Ave.
5	3	4	12/08/2024	899	3	John	Ken	2st Link
6	4	4	12/08/2024	899	4	Ivy	Tan	67th St.

# Method 2 - Index-Based Nested-Loop

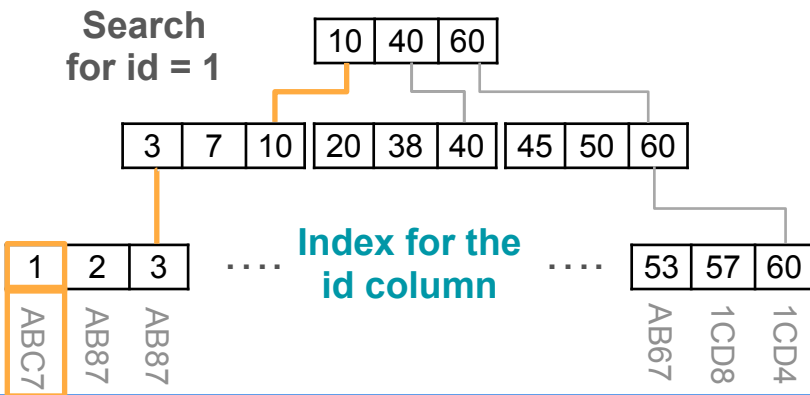
Can be used when an index exists for one of the join attributes

orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.



# Method 2 - Index-Based Nested-Loop

Can be used when an index exists for one of the join attributes

## orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

## customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

## Combined Table

order_id	customer_id	product_id	date_time	purchase_amount	id	first_name	last_name	address
1	1	1	12/08/2024	700	1	Jane	Doe	74th St
2	1	2	12/08/2024	99	1	Jane	Doe	74th St
3	1	3	12/08/2024	100	1	Jane	Doe	74th St
4	2	4	12/08/2024	899	2	Mary	Ann	19th Ave.
5	3	4	12/08/2024	899	3	John	Ken	2st Link
6	4	4	12/08/2024	899	4	Ivy	Tan	67th St.

# Method 3 - Hash-Join

## orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

## customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

Customer ID



Bucket ID

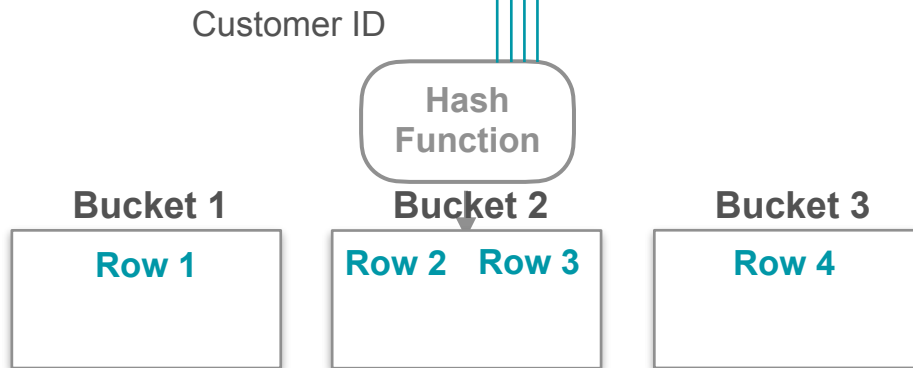
# Method 3 - Hash-Join

## orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

## customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.





# Method 3 - Hash-Join

## orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

## customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

Customer ID

Hash  
Function

Bucket 1

Row 1  
Row 1 Row 2  
Row 3

Bucket 2

Row 2 Row 3  
Row 4 Row 5

Bucket 3

Row 4  
Row 6

# Method 3 - Hash-Join

orders

order id	customer id	product id	date time	purchase amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

customers

id	first name	last name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

Bucket 1

Row 1
Row 1 Row 2
Row 3

Bucket 2

Row 2 Row 3
Row 4 Row 5

Bucket 3

Row 4
Row 6

# Method 3 - Hash-Join

## orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

## customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

### Bucket 1

Row 1  
Row 1 Row 2  
Row 3

### Bucket 2

Row 2 Row 3  
Row 4 Row 5

### Bucket 3

Row 4  
Row 6

# Method 3 - Hash-Join

## orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

## customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

### Bucket 1

Row 1
Row 1   Row 2
Row 3

### Bucket 2

Row 2   Row 3
Row 4   Row 5

### Bucket 3

Row 4
Row 6

# Method 3 - Hash-Join

## orders

order_id	customer_id	product_id	date_time	purchase_amount
1	1	1	12/08/2024	700
2	1	2	12/08/2024	99
3	1	3	12/08/2024	100
4	2	4	12/08/2024	899
5	3	4	12/08/2024	899
6	4	4	12/08/2024	899

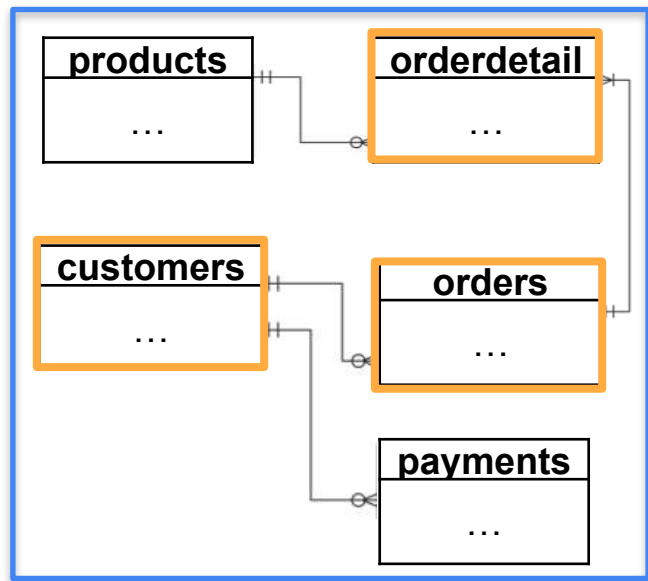
## customers

id	first_name	last_name	address
1	Jane	Doe	74th St
2	Mary	Ann	19th Ave.
3	John	Ken	2st Link
4	Ivy	Tan	67th St.

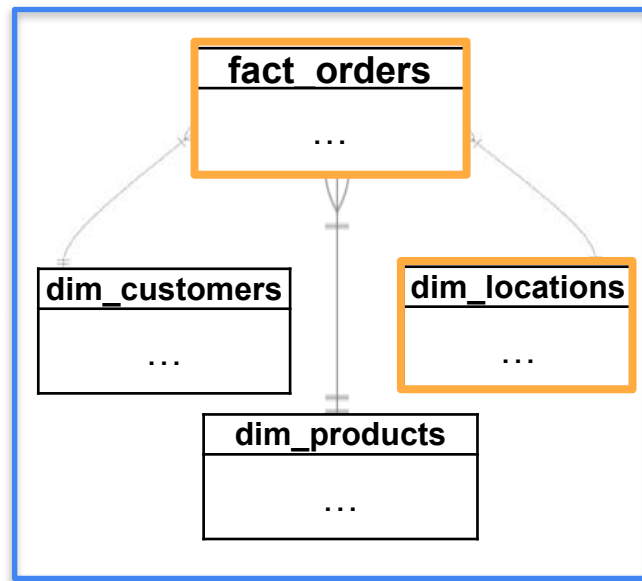
## Combined Table

order_id	customer_id	product_id	date_time	purchase_amount	id	first_name	last_name	address
1	1	1	12/08/2024	700	1	Jane	Doe	74th St
2	1	2	12/08/2024	99	1	Jane	Doe	74th St
3	1	3	12/08/2024	100	1	Jane	Doe	74th St
4	2	4	12/08/2024	899	2	Mary	Ann	19th Ave.
5	3	4	12/08/2024	899	3	John	Ken	2st Link
6	4	4	12/08/2024	899	4	Ivy	Tan	67th St.

# Schemas and Joins



Normalized Schema



Star Schema



# Schemas and Joins

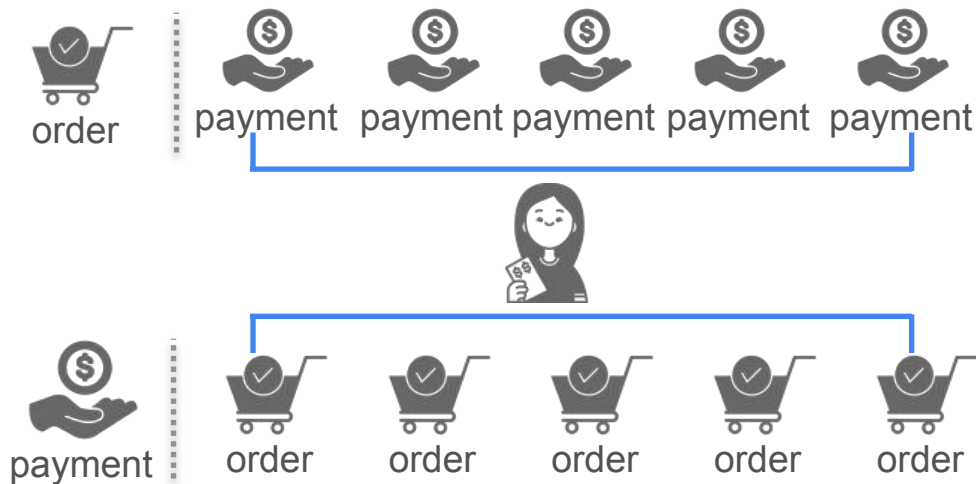
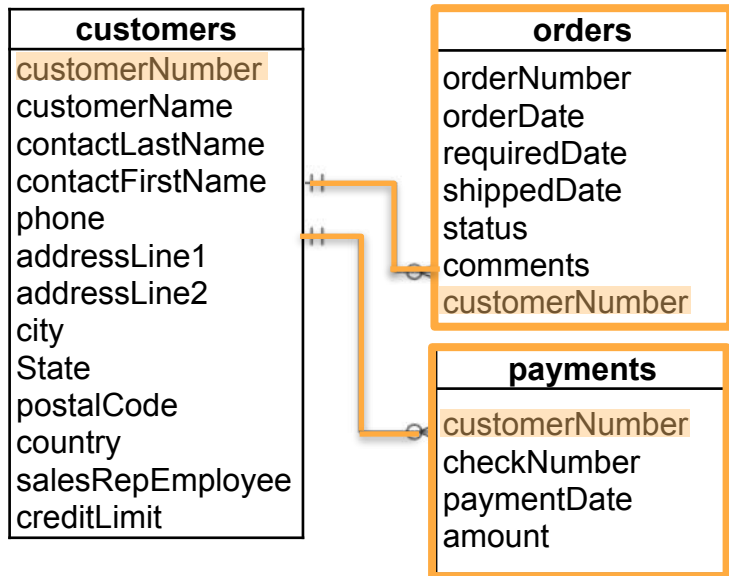
Customers info	....	Orders info	....	Payments info	....	Products info	....



No joins to perform

**One Big Table (OBT)**

# Many-to-Many Relationships



## Normalized Schema



# Many-to-Many Relationships

customers
customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
State
postalCode
country
salesRepEmployee
creditLimit

orders
orderNumber
orderDate
requiredDate
shippedDate
status
comments
customerNumber

payments
customerNumber
checkNumber
paymentDate
amount

Payments

payment_date	customer Number	amount
12/1/2024	1	100
12/2/2024	1	23
1/1/2025	1	597
2/1/2025	1	89
3/1/2025	1	44

Orders

order_date	customer Number	order_number
12/1/2024	1	789VT
12/2/2024	1	786UI
3/1/2025	1	597AB
4/1/2025	1	898VB
5/1/2025	1	131MM

25 row outputs

payment_date	customer Number	amount	order_date	customer Number	order_number
12/1/2024	1	100	12/1/2024	1	789VT
12/1/2024	1	100	12/2/2024	1	786UI
12/1/2024	1	100	3/1/2025	1	597AB
12/1/2024	1	100	4/1/2025	1	898VB
12/2/2024	1	23	12/1/2024	1	789VT
12/2/2024	1	23	12/2/2024	1	786UI
12/2/2024	1	23	3/1/2025	1	597AB

.....

.....

Normalized Schema

# Many-to-Many Relationships

## Row Explosion

When a query returns more rows than what is anticipated

- Check your query to see if it correctly describes what you intended the join to do
- Add a table that correctly maps payment to orderNumber

Payments

payment_date	customer Number	amount
12/1/2024	1	100
12/2/2024	1	23
1/1/2025	1	597
2/1/2025	1	89
3/1/2025	1	44

Orders

order_date	customer Number	order_number
12/1/2024	1	789VT
12/2/2024	1	786UI
3/1/2025	1	597AB
4/1/2025	1	898VB
5/1/2025	1	131MM

25 row outputs

payment_date	customer Number	amount	order_date	customer Number	order_number
12/1/2024	1	100	12/1/2024	1	789VT
12/1/2024	1	100	12/2/2024	1	786UI
12/1/2024	1	100	3/1/2025	1	597AB
12/1/2024	1	100	4/1/2025	1	898VB
12/2/2024	1	23	12/1/2024	1	789VT
12/2/2024	1	23	12/2/2024	1	786UI
12/2/2024	1	23	3/1/2025	1	597AB

.....

.....



DeepLearning.AI

# Queries

---

## **The Aggregate Queries**

# Aggregating Queries

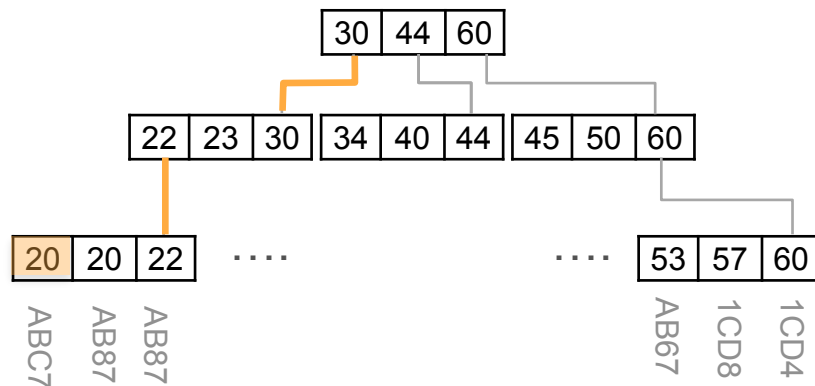
```
SELECT MIN(price) FROM orders
```

## Full Table Scan

Order ID	Price	Product SKU	Quantity	Customer ID	Store ID	Country
1	40	458650	10	67t	3	Canada
2	25	902348	14	56t	3	Canada
3	45	1255893	12	87q	4	Canada
4	50	456829	13	98q	1	USA
5	34	568298	12	98q	1	USA
6	44	563783	4	67t	1	USA
7	22	234589	5	56u	2	Brazil
8	30	267895	12	78y	3	Canada
9	60	545659	14	13t	5	Mexico

...

## Use the Index (if available)



# Aggregating Queries with GROUP BY

```
SELECT MIN(price) FROM orders GROUP BY country
```

Order ID	Price	Product SKU	Quantity	Customer ID	Store ID	Country
1	40	458650	10	67t	3	Canada
2	23	902348	14	56t	3	Canada
3	45	1255893	12	87q	4	Canada
4	50	456829	13	98q	1	USA
5	34	568298	12	98q	1	USA
6	44	563783	4	67t	1	USA
7	22	234589	5	56u	2	Brazil
8	33	267895	12	78y	3	Canada
9	60	545659	14	13t	5	Mexico
...						

- Partitioning can be done using a sorting algorithm or hash function
- Or you can use an index to group the rows

# Aggregating Queries - Row VS Columnar Storage

```
SELECT MIN(price) FROM orders
```

Order ID	Price	Product SKU	Quantity	Customer ID
1	40	458650	10	67t
2	23	902348	14	56t
3	45	125589	12	87q
4	50	456829	13	98q

**Row Storage** — Need to transfer all rows from disk to memory

1	40	45865	10	67t	2	23	90234	14	56t	3	45	125589	12	87q	...
---	----	-------	----	-----	---	----	-------	----	-----	---	----	--------	----	-----	-----

**Columnar Storage** — Transfer only the relevant columns from disk to memory

1	2	3	4	40	23	45	50	458650	902348	125589	456829	...
---	---	---	---	----	----	----	----	--------	--------	--------	--------	-----



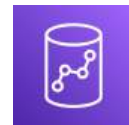
DeepLearning.AI

# Queries

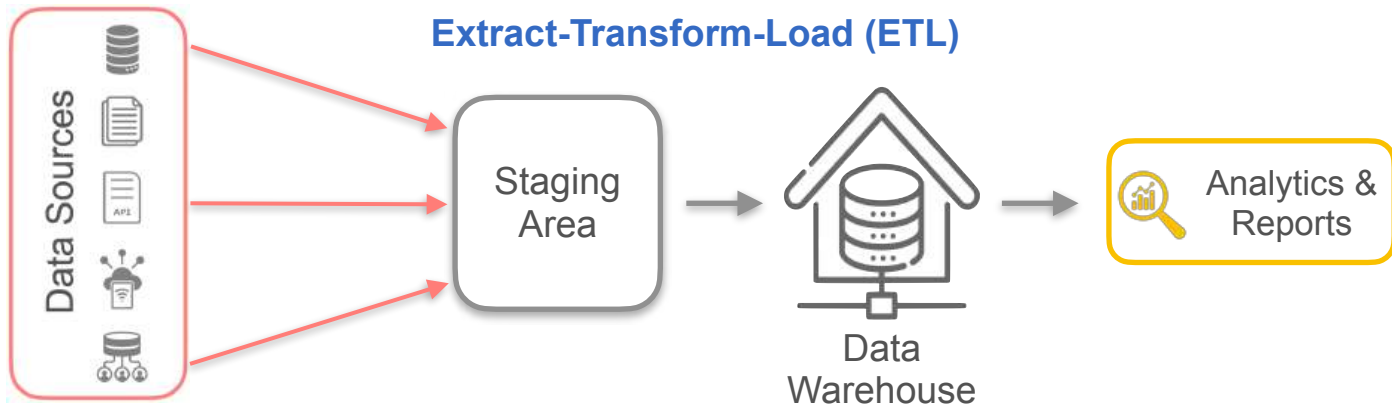
---

## **Amazon Redshift Cloud Data Warehouse**

# Amazon Redshift Cloud Data Warehouse

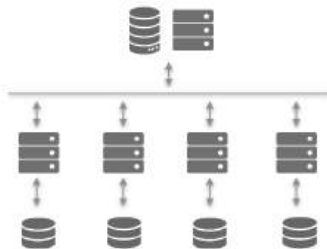


Amazon Redshift

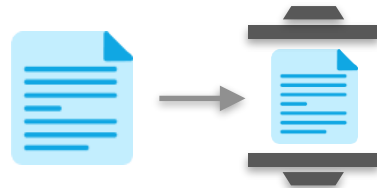


Order ID	Price	Product SKU	Quantity	Customer ID
1	40	45865	10	67t
2	23	90234	14	56t
3	45	12558	12	87q
4	50	45882	13	98q
...	...	...	...	...

Columnar Data Storage



Massively Parallel Processing (MPP)



Data Compression



# Amazon Redshift - Column Storage

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
4	50	08-23-2024	13	2749
...	...	...	...	...



Stores data  
Column by column

## Physical Storage

bytes representing 1st column	bytes representing 2nd column	bytes representing 3rd column	...
-------------------------------	-------------------------------	-------------------------------	-----

# Amazon Redshift - Column Storage

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
4	50	08-23-2024	13	2749
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...



- Analytical Queries
- OLAP Workloads

bytes representing 1st column	bytes representing 2nd column	bytes representing 3rd column	...
-------------------------------	-------------------------------	-------------------------------	-----

# Amazon Redshift - Column Storage

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
4	50	08-23-2024	13	2749
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...



- Analytical Queries
- OLAP Workloads



Cost Efficiency



Speeding Up  
Query Performance



Data  
Warehousing

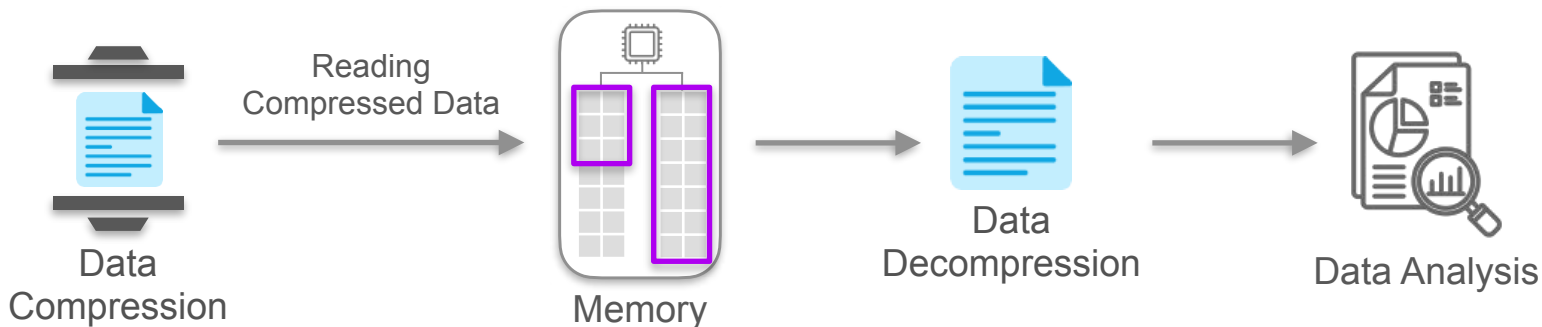
bytes representing 1st column	bytes representing 2nd column	bytes representing 3rd column	...
-------------------------------	-------------------------------	-------------------------------	-----

# Amazon Redshift - Data Compression

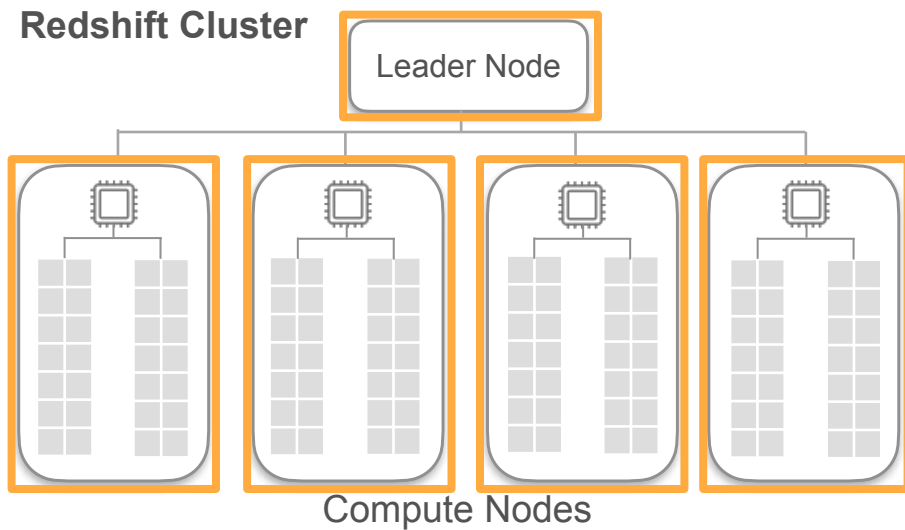
Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
4	50	08-23-2024	13	2749
...	...	...	...	...

Columnar Storage

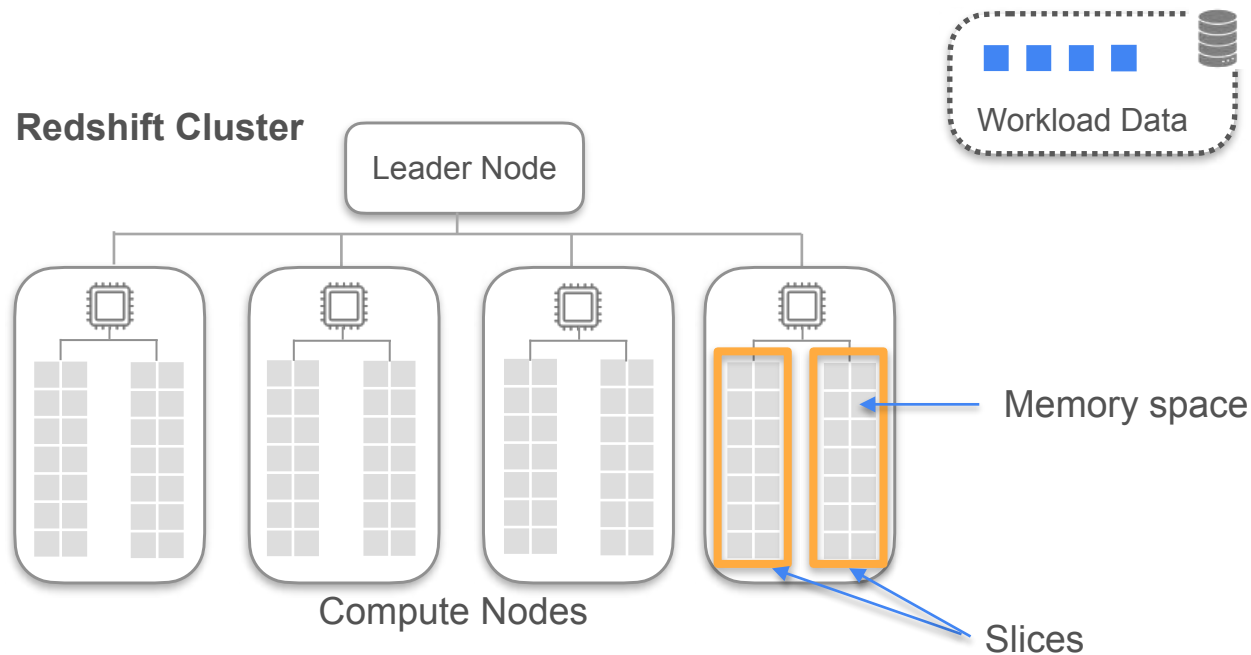
- Save storage space
- Reading less data from disk



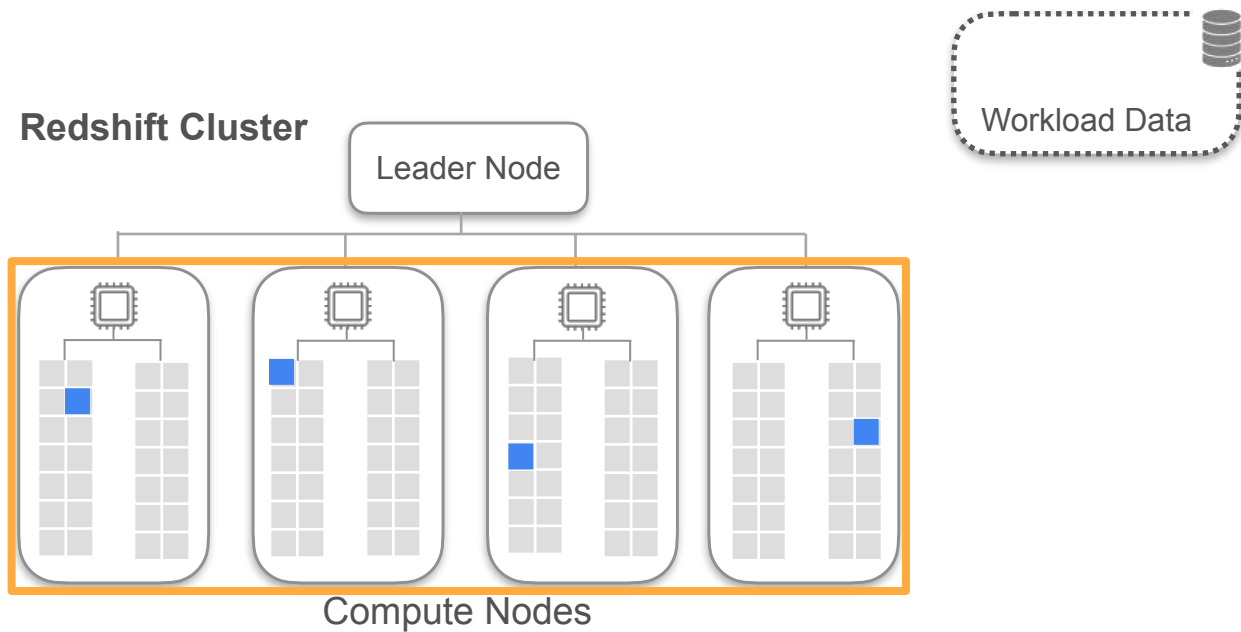
# Amazon Redshift - Massive Parallel Processing



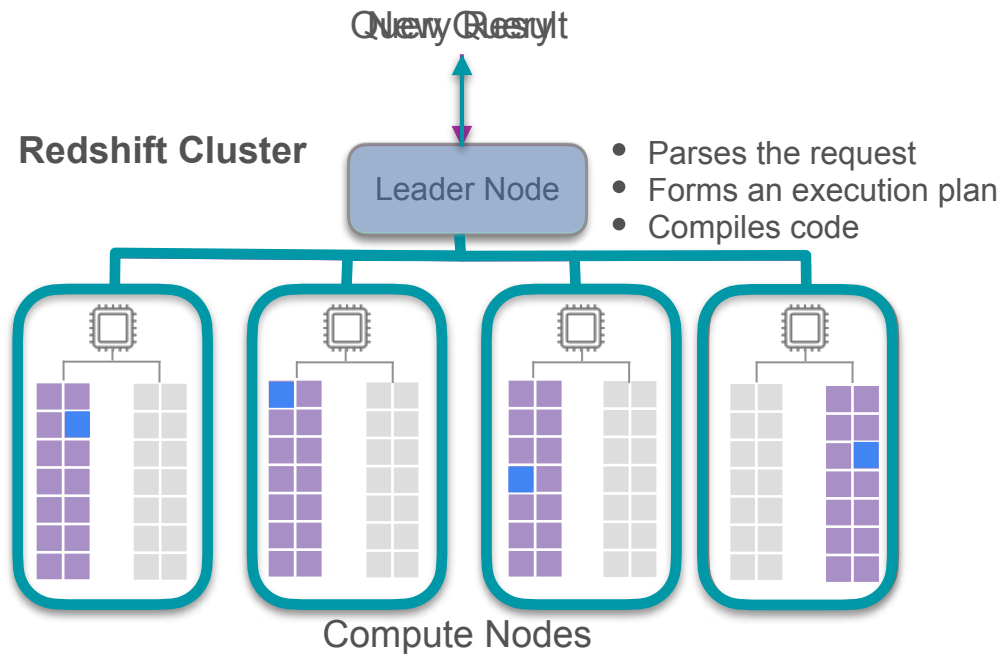
# Amazon Redshift - Massive Parallel Processing



# Amazon Redshift - Massive Parallel Processing

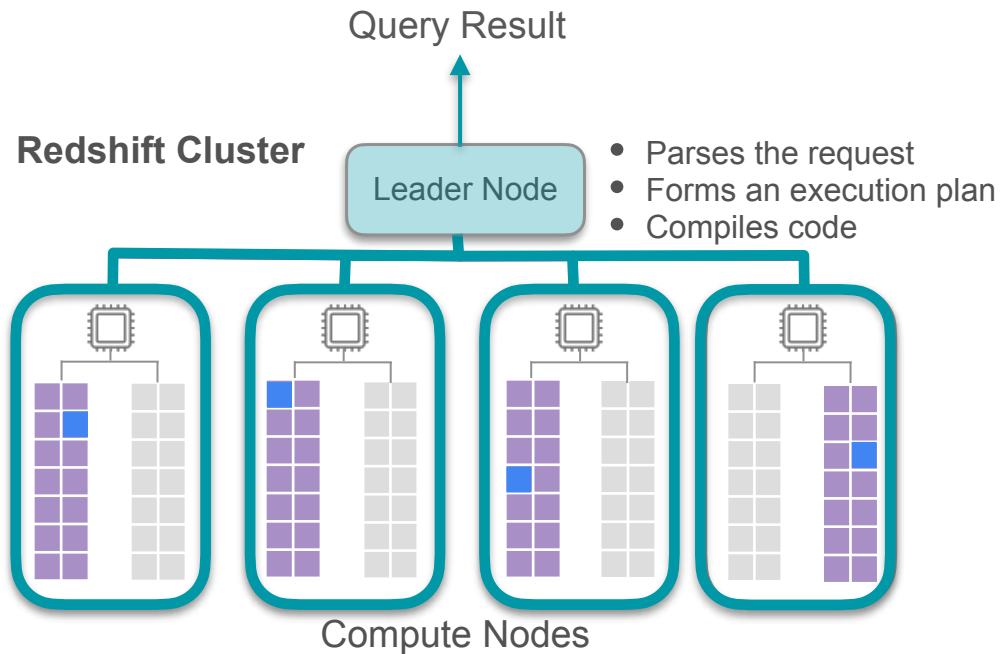


# Amazon Redshift - Massive Parallel Processing





# Amazon Redshift - Massive Parallel Processing



## Query Performance

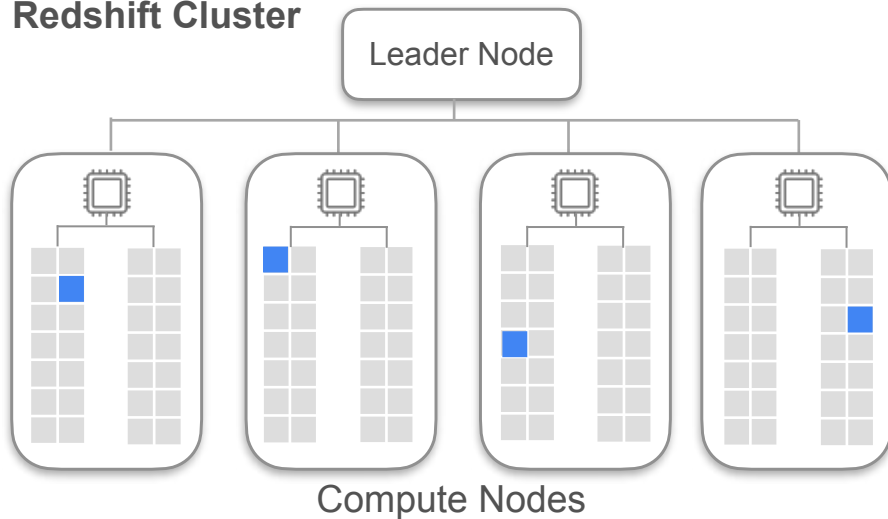
- Number of nodes
- Type of nodes

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Distribution Style

### Redshift Cluster



---

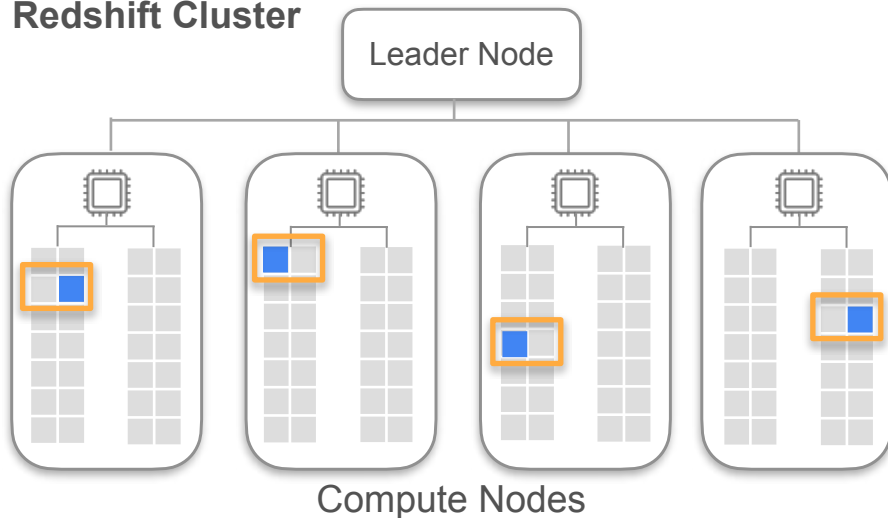
## Sort Key

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Distribution Style

### Redshift Cluster



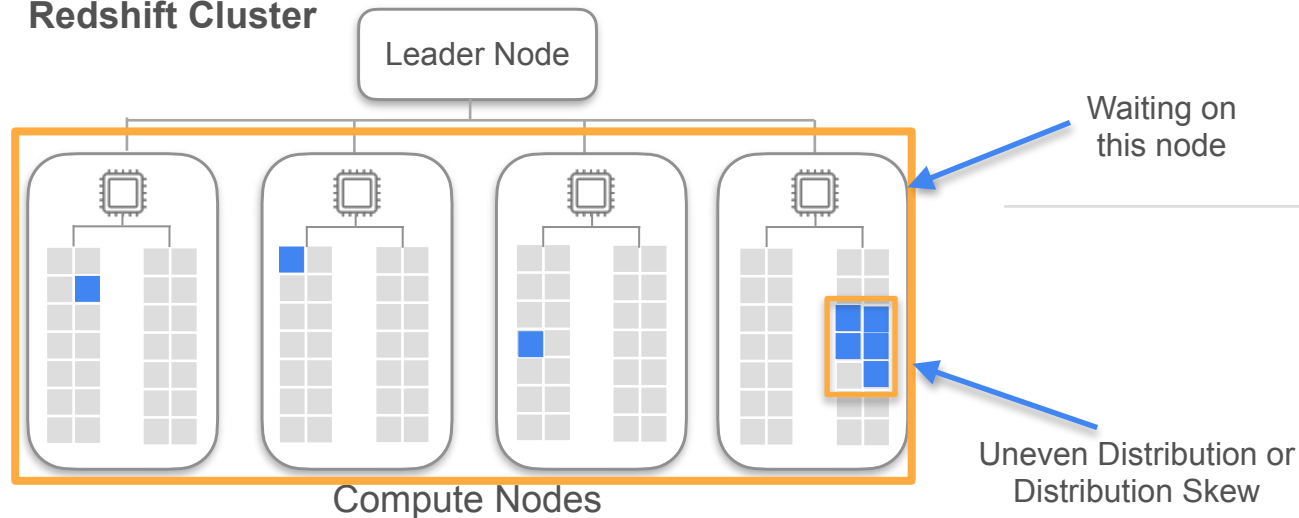
# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Distribution Style

1. Uniform Distribution across nodes

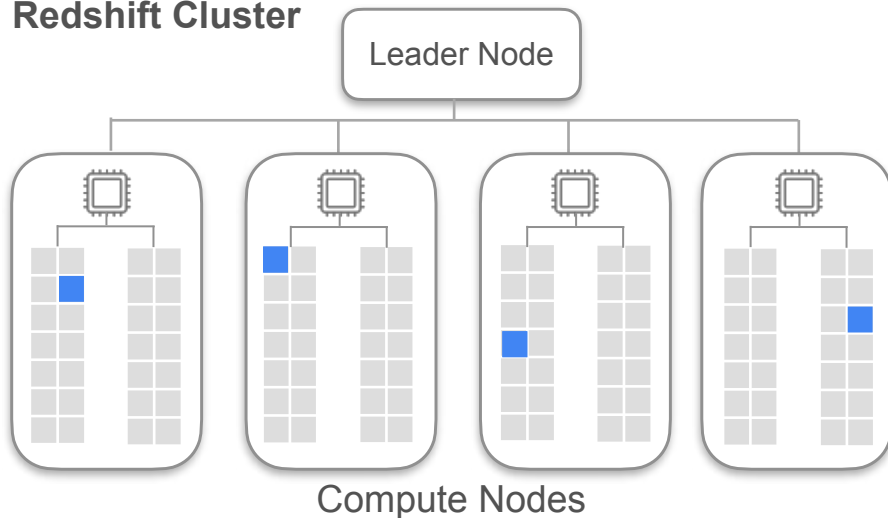
### Redshift Cluster



# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



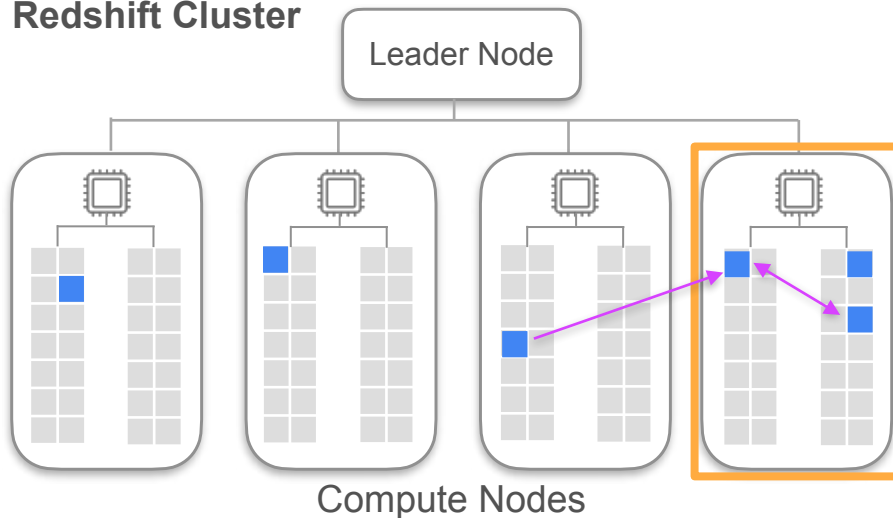
## Distribution Style

1. Uniform Distribution across nodes
2. Minimize data movement across nodes  
*(Impact query performance and cost)*

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



## Distribution Style

1. Uniform Distribution across nodes
2. Minimize data movement across nodes  
*(Impact query performance and cost)*

**AUTO**

**EVEN**

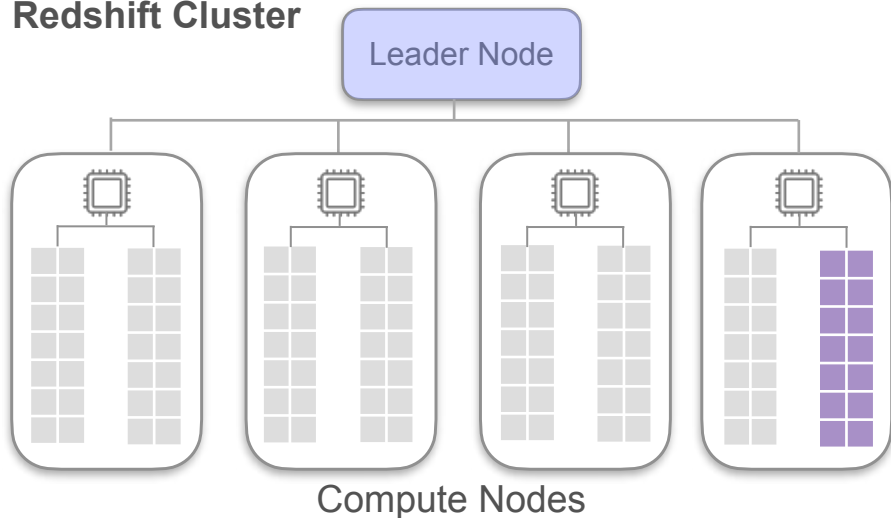
**KEY**

**ALL**

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



## Distribution Style

1. Uniform Distribution across nodes
2. Minimize data movement across nodes  
*(Impact query performance and cost)*

**AUTO**

**EVEN**

**KEY**

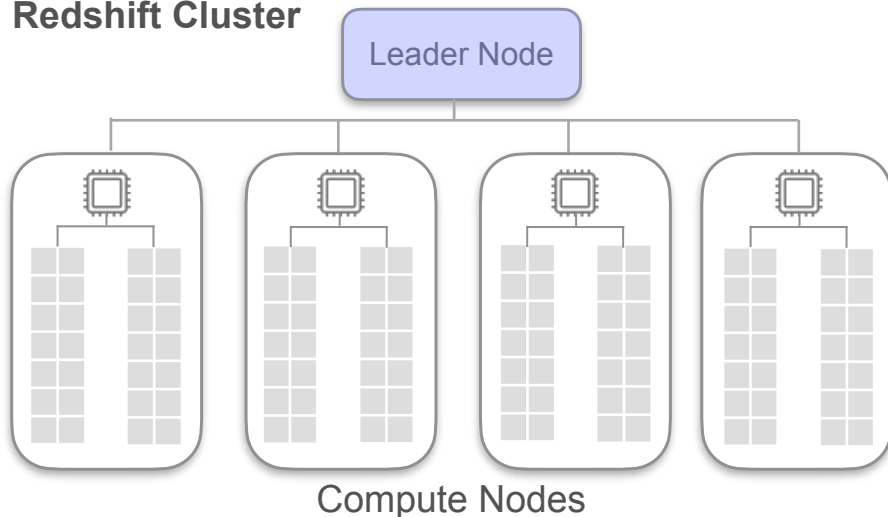
- Distribute rows based on specified column

**ALL**

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



## Distribution Style

1. Uniform Distribution across nodes
2. Minimize data movement across nodes  
*(Impact query performance and cost)*

**AUTO** (ALL / EVEN / KEY)

**EVEN**

- Default distribution style

**KEY**

**ALL**

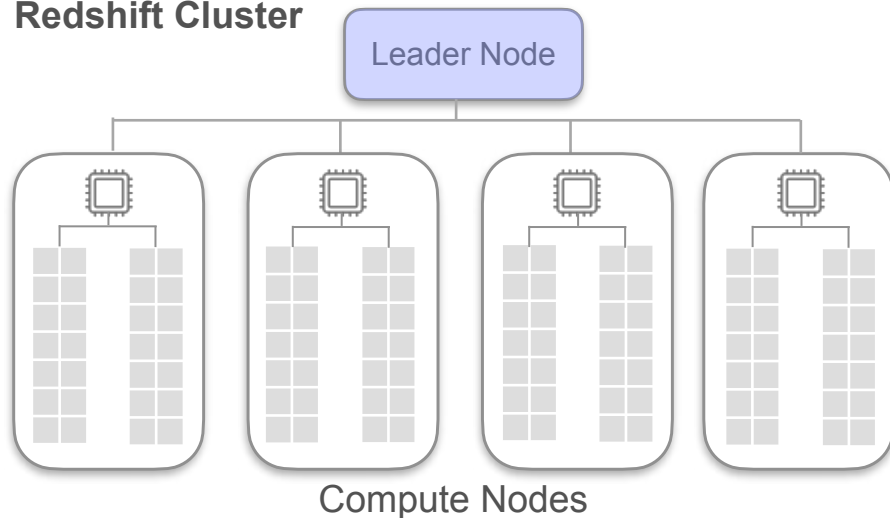
- Distribute rows based on specified column



# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



## Distribution Style

1. Uniform Distribution across nodes
2. Minimize data movement across nodes  
*(Impact query performance and cost)*

### AUTO (ALL / EVEN / KEY)

- Default distribution style

### EVEN

- Round-robin distribution
- No joins

### KEY

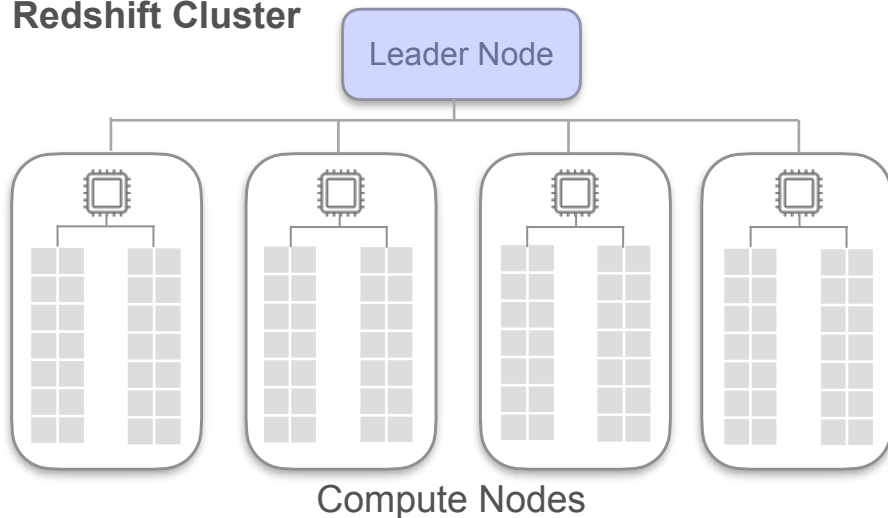
- Distribute rows based on specified column

### ALL

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



## Distribution Style

1. Uniform Distribution across nodes
2. Minimize data movement across nodes  
*(Impact query performance and cost)*

### AUTO (ALL / EVEN / KEY)

- Default distribution style

### EVEN

- Round-robin distribution
- No joins

### KEY

- Distribute rows based on specified column

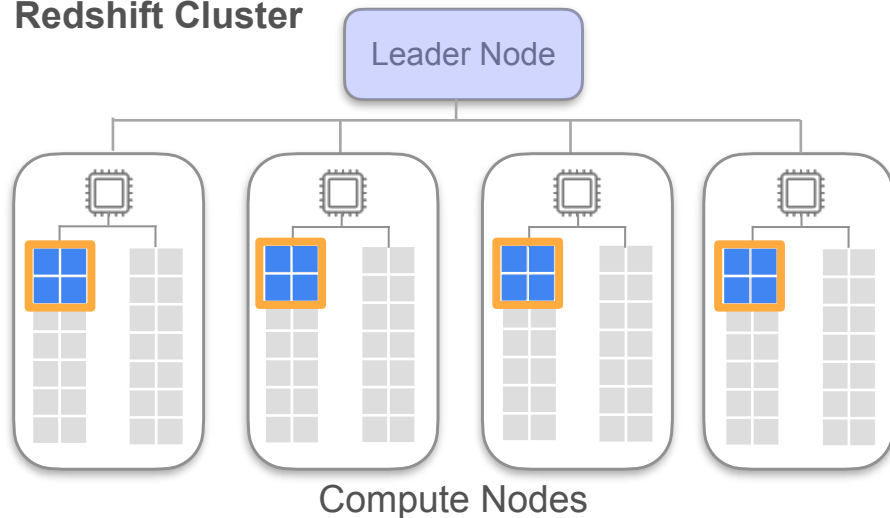
### ALL

- Copies table to each node

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



## Distribution Style

1. Uniform Distribution across nodes
2. Minimize data movement across nodes  
*(Impact query performance and cost)*

### **AUTO** (ALL / EVEN / KEY)

- Default distribution style

### **EVEN**

- Round-robin distribution
- No joins

### **KEY**

- Distribute rows based on specified column

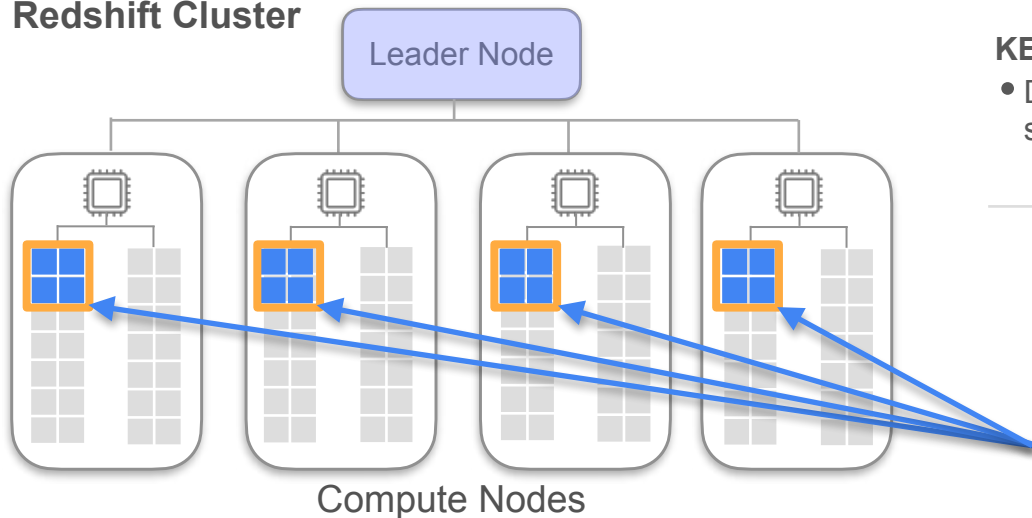
### **ALL**

- Copies table to each node
- Eliminates data shuffling

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



## Distribution Style

1. Uniform Distribution across nodes
2. Minimize data movement across nodes  
*(Impact query performance and cost)*

### AUTO (ALL / EVEN / KEY)

- Default distribution style

### EVEN

- Round-robin distribution
- No joins

### KEY

- Distribute rows based on specified column

### ALL

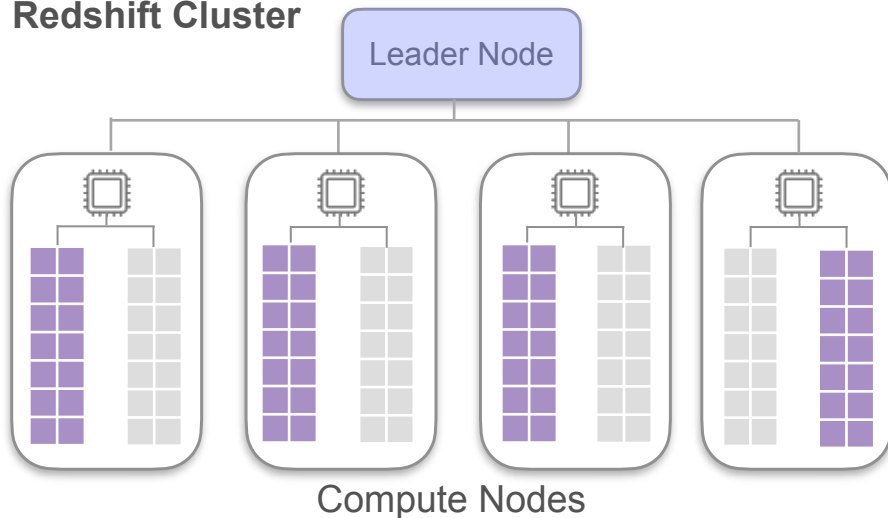
- Copies table to each node
- Eliminates data shuffling
- Takes longer

Multiples the storage required by the number of nodes

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



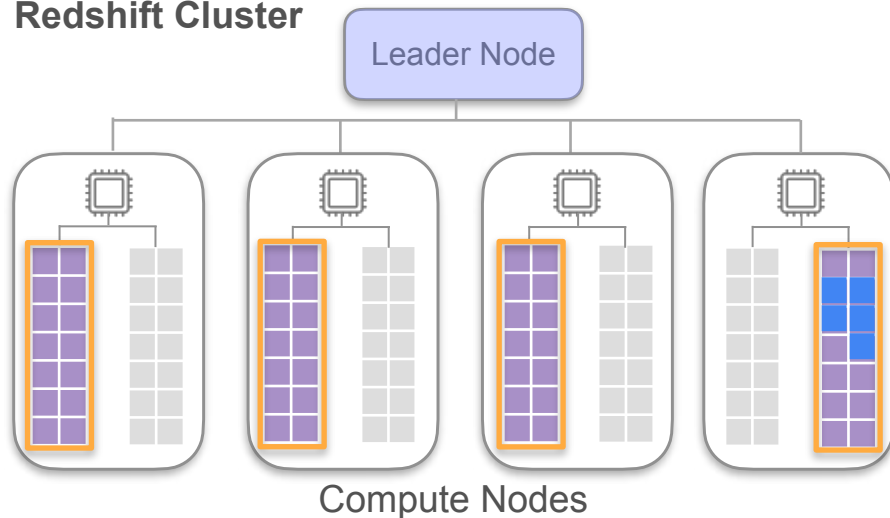
## Sort Key

- Stores data on disk based on sort key
- Helps query optimizer determine optimal query plan

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



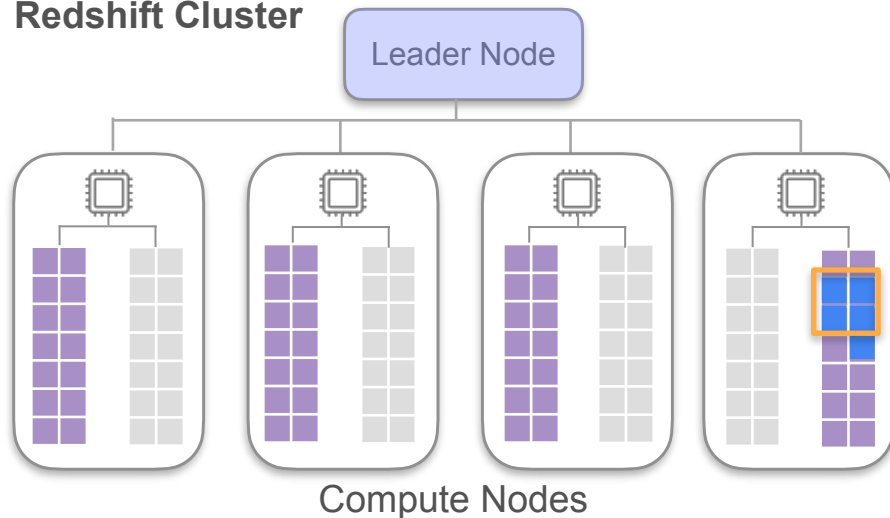
## Sort Key

- Stores data on disk based on sort key
- Helps query optimizer determine optimal query plan
- Minimizes disk read operations
- Speed up query

# Table Design

Order ID	Price	Order date	Quantity	Customer ID
1	40	05-22-2025	10	1337
2	23	06-15-2024	14	124
3	45	07-03-2024	12	3465
...	...	...	...	...

## Redshift Cluster



## Sort Key (Similar to OLTP database indexes)

- Stores data on disk based on sort key
- Helps query optimizer determine optimal query plan
- Minimizes disk read operations
- Speed up query



DeepLearning.AI

## Lab Walkthrough

---

# Query Comparison Between Row and Columnar Databases



# Lab Overview

- Compare their execution times.
- In this video: overview of the dataset and the results of the experiments.

## Row-based Database



Analytical queries



Update/Delete queries

---

## Column-based Database



Analytical queries

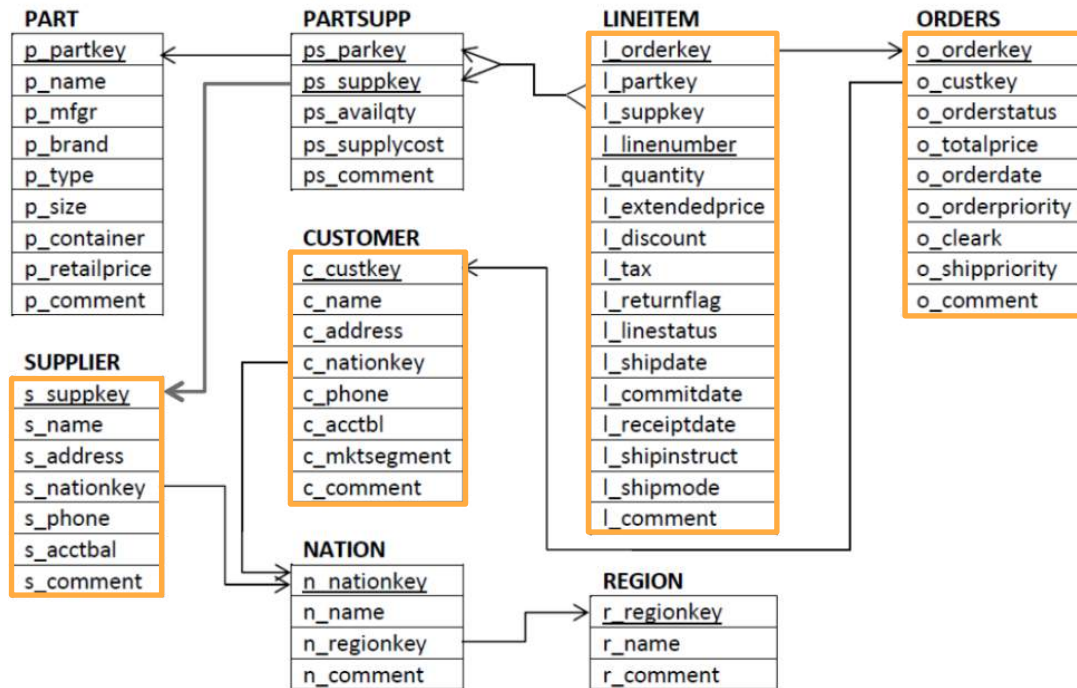


Update/Delete queries

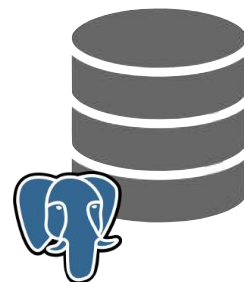
# Benchmarking Dataset

## TPC-H Benchmark:

Evaluate the performance of various database systems



## Row-based Database



PostgreSQL

5 Analytical queries

## Column-based Database



Amazon Redshift

5 Analytical queries

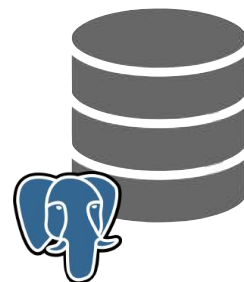
# Benchmarking Dataset

Generate 50 rows containing random entries:

- Write the rows and compare the execution time
- Delete the rows and compare the execution time

LINEITEM
<u>l_orderkey</u>
l_partkey
l_suppkey
<u>l_linenumber</u>
l_quantity
l_extendedprice
l_discount
l_tax
l_returnflag
l_linestatus
l_shipdate
l_commitdate
l_receiptdate
l_shipinstruct
l_shipmode
l_comment

## Row-based Database



PostgreSQL



Update/Delete queries

## Column-based Database



Amazon Redshift



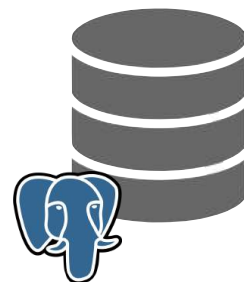
Update/Delete queries

# Lab Overview

You will first:

- Establish a connection to the redshift database
- Issue some queries

## Row-based Database



PostgreSQL



## Column-based Database



Amazon Redshift



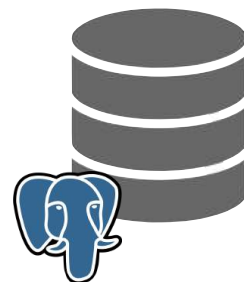
# Experiments

## TPC-H Benchmark

Execution times for the **analytical queries** run once

	Column-Based Database	Row-Based Database
First TPC-H query	291 ms	2 min 41 s
Second TPC-H query	87 ms	1 min 10 s
Third TPC-H query	4.33 s	5 min 35 s
Fourth TPC-H query	233 ms	1 min 16 s
Fifth TPC-H query	3.1 s	1 min 36 s

## Row-based Database



PostgreSQL

5 Analytical queries

## Column-based Database



Amazon Redshift

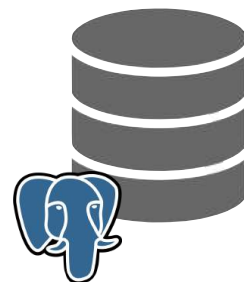
5 Analytical queries

# Experiments

Execution times for the **write/delete queries** run once

	Column-Based Database	Row-Based Database
Write 50 rows	7.4 s	456 ms
Delete 50 rows	16.4 s	176 ms

## Row-based Database



PostgreSQL



Update/Delete queries

## Column-based Database



Amazon Redshift



Update/Delete queries



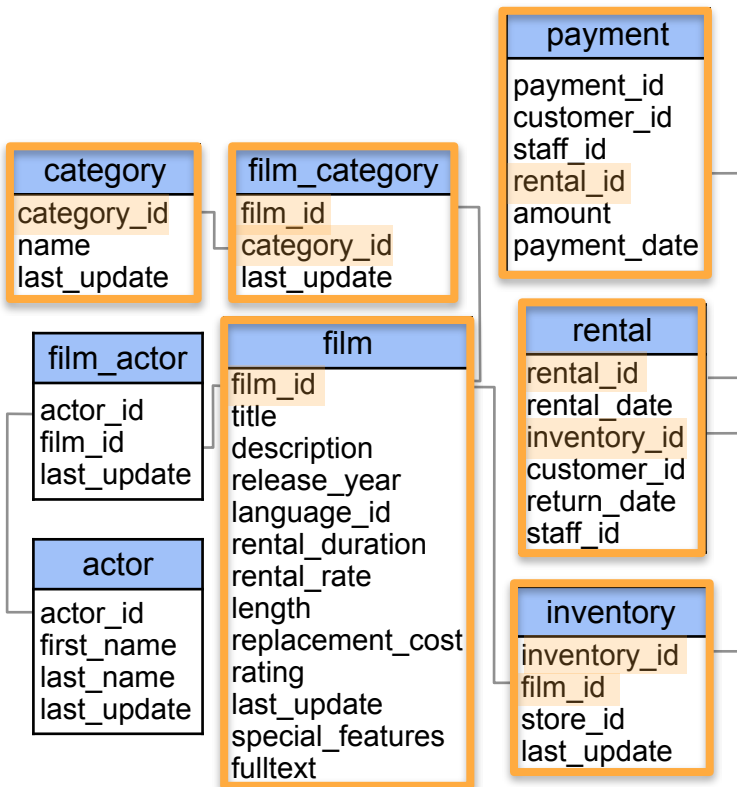
DeepLearning.AI

# Queries

---

## **Additional Query Strategies**

# Leverage Query Caching



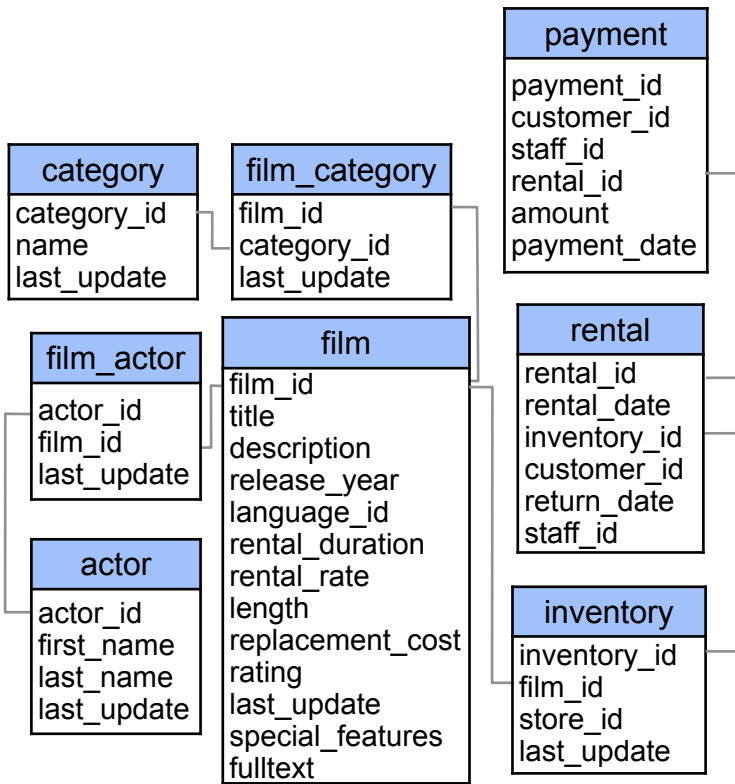
Total amount spent: family, drama, and comedy categories

```
SELECT SUM(payment.amount) AS amount
FROM payment
JOIN rental ON rental.rental_id = payment.rental_id
JOIN inventory ON inventory.inventory_id = rental.inventory_id
JOIN film ON film.film_id = inventory.film_id
JOIN film_category ON film_category.film_id = film.film_id
JOIN category ON category.category_id = film_category.category_id
WHERE category.name IN ('Family', 'Drama', 'Comedy')
GROUP BY category.name
ORDER BY amount
```

- Running a complex query frequently can be costly
- Many databases allow you to cache query results
- **Query caching** can reduce the load on your database and enhance the user experience



# Prioritize Readability



## Readable queries:

- Less likely to contain errors
- Simpler to debug
- Easier to collaborate on

### CTE (Common Table Expressions)

Creates a temporary result set that you can reference in your query

```
WITH selected_film AS (  
    SELECT film_id  
    FROM film  
    WHERE title = "Rocky War"  
),  
film_actors_id AS (  
    SELECT actor_id  
    FROM film_actor  
    WHERE film_id IN selected_film  
)  
SELECT actor.first_name, actor.last_name  
FROM actor  
WHERE actor_id IN film_actors_id
```

# Prioritize Readability

## Nested Subqueries

```
SELECT actor.first_name, actor.last_name
FROM actor
WHERE actor.actor_id IN (
    SELECT actor_id
    FROM film_actor
    WHERE film_id = (
        SELECT film_id
        FROM film
        WHERE title = 'Rocky War'
    )
)
```

## Readable queries:

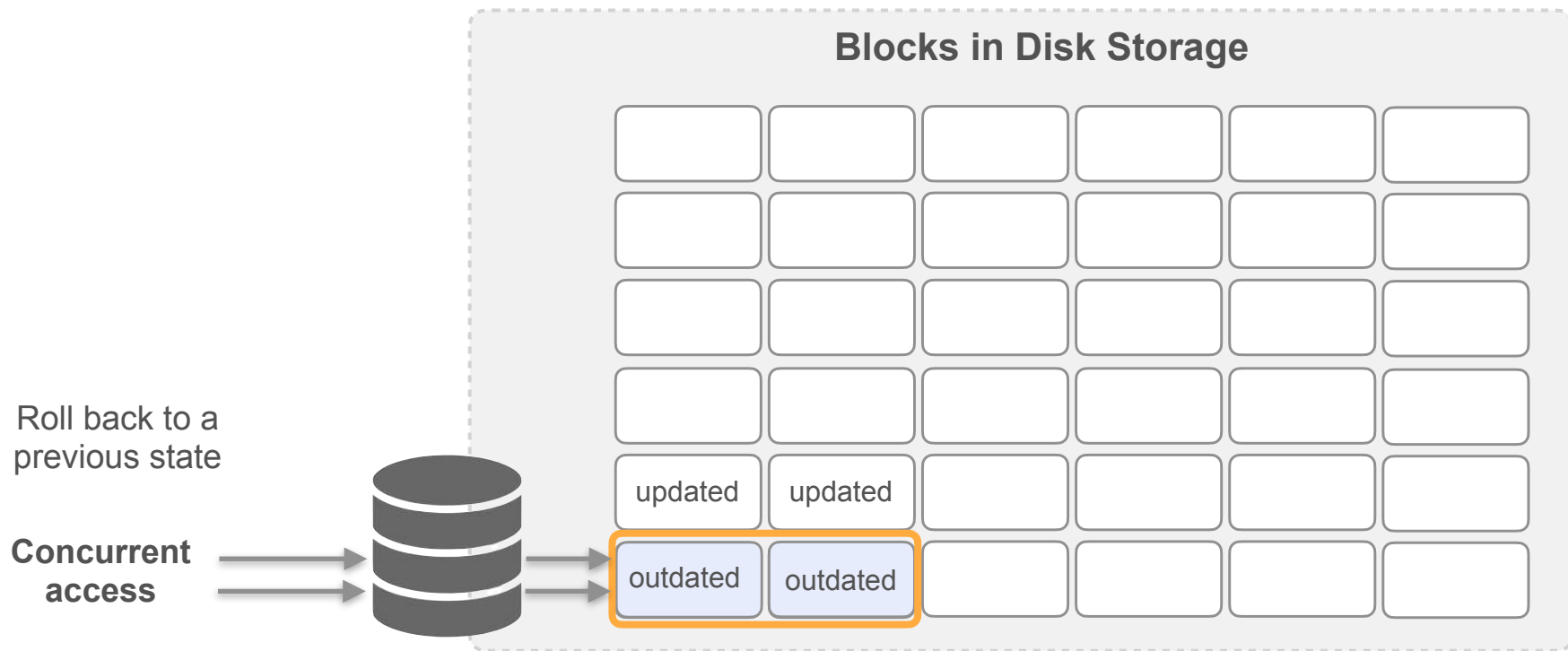
- Less likely to contain errors
- Simpler to debug
- Easier to collaborate on

## CTE (Common Table Expressions)

Creates a temporary result set that you can reference in your query

```
WITH selected_film AS (
    SELECT film_id
    FROM film
    WHERE title = "Rocky War"
),
film_actors_id AS (
    SELECT actor_id
    FROM film_actor
    WHERE film_id IN selected_film
)
SELECT actor.first_name, actor.last_name
FROM actor
WHERE actor_id IN film_actors_id
```

# Database Resources



# Database Resources

## Table Bloat

The data size on the disk exceeds the actual data size

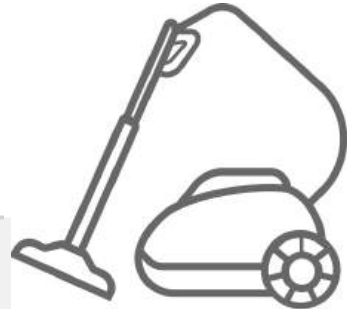
- Wasted disk space
- Slow queries
- Suboptimal and inaccurate execution plans
- Inefficient indexes



## Blocks in Disk Storage

updated	outdated			updated	updated
outdated	updated	outdated	outdated	outdated	outdated
outdated	outdated	updated	updated		outdated
outdated	outdated	outdated	outdated	updated	outdated
updated	updated	outdated	outdated	outdated	outdated
outdated	outdated	updated	updated	updated	updated

# Database Resources



## Vacuuming

Removing the dead records

- Critical for relational databases (PostgreSQL, MySQL)
- Familiarize yourself with the details of vacuuming



## Blocks in Disk Storage

updated	outdated			updated	updated
outdated	updated	outdated	outdated	outdated	outdated
outdated	outdated	updated	updated		outdated
outdated	outdated	outdated	outdated	updated	outdated
updated	updated	outdated	outdated	outdated	outdated
outdated	outdated	updated	updated	updated	updated



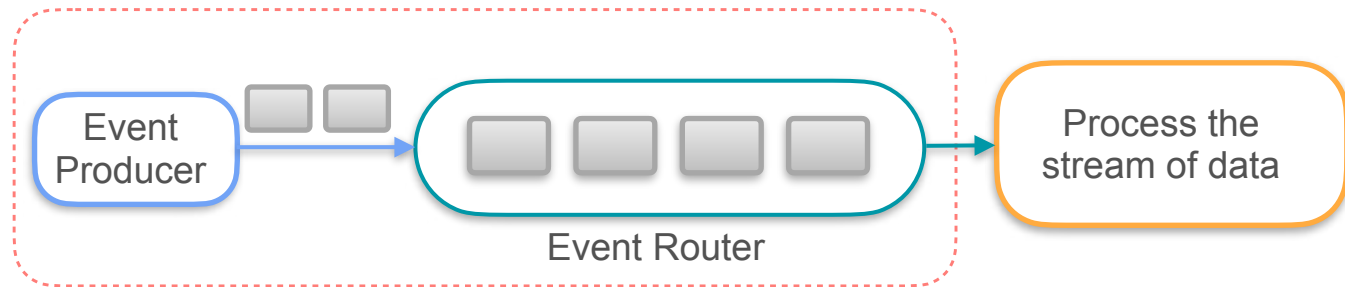
DeepLearning.AI

# Queries

---

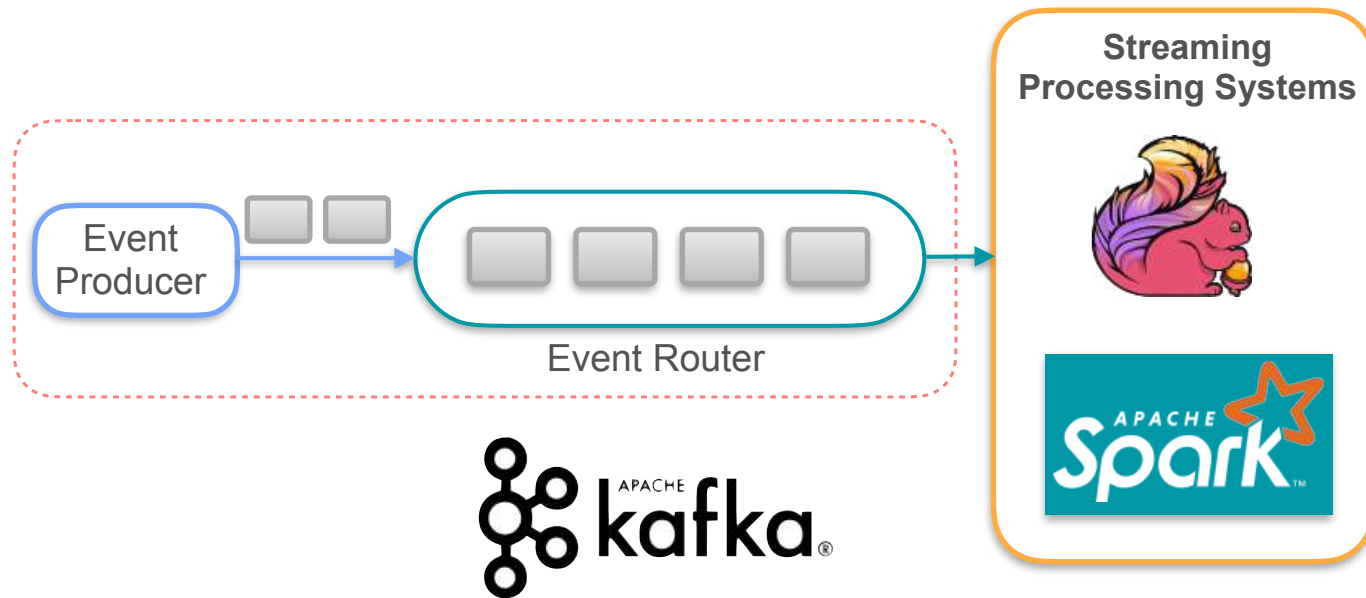
## Queries on Streaming Data

# Streaming System



# Streaming Processing System

Apply SQL queries continuously on streaming data





# Queries on Streaming Data

## Windowed Query

Bound your queries using a window & apply operations over that window.  
(e.g. *aggregating, adding, removing* data)

**Session Window**

**Fixed-time Window**

**Sliding Window**

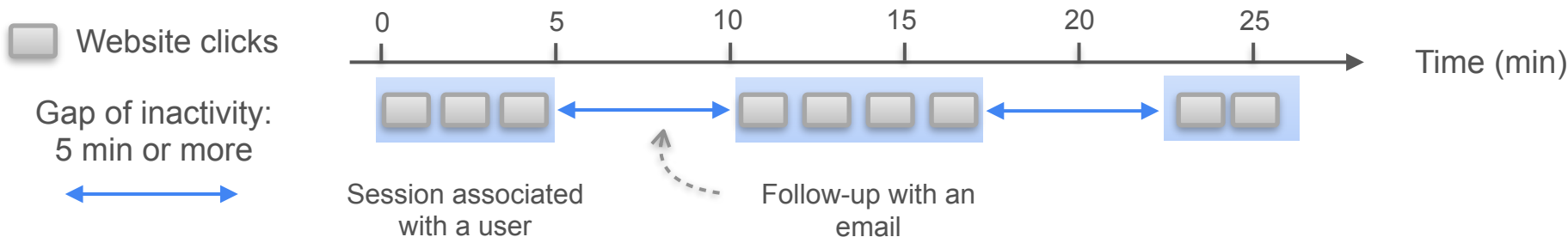
---

# Queries on Streaming Data

## Windowed Query

Bound your queries using a window & apply operations over that window.  
(e.g. *aggregating, adding, removing* data)

### Session Window

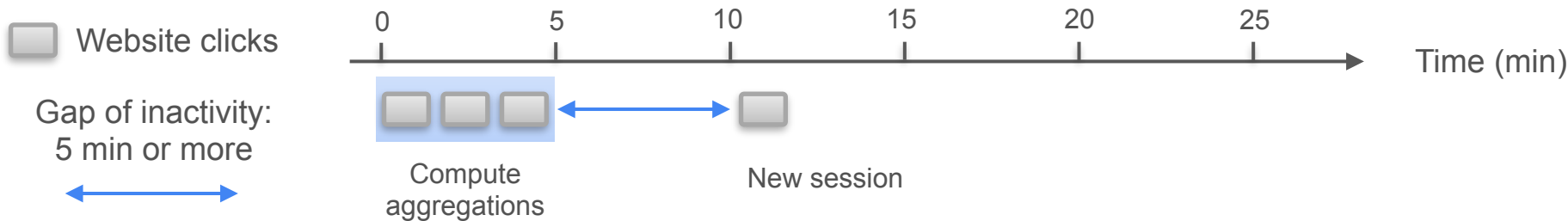


# Queries on Streaming Data

## Windowed Query

Bound your queries using a window & apply operations over that window.  
(e.g. *aggregating, adding, removing* data)

### Session Window



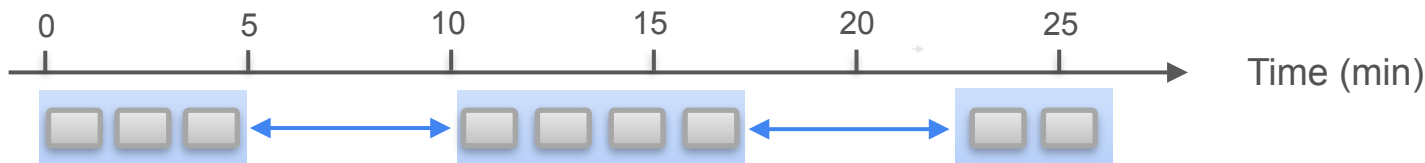
# Queries on Streaming Data

## Windowed Query

Bound your queries using a window & apply operations over that window.  
(e.g. *aggregating, adding, removing* data)

### Session Window

---

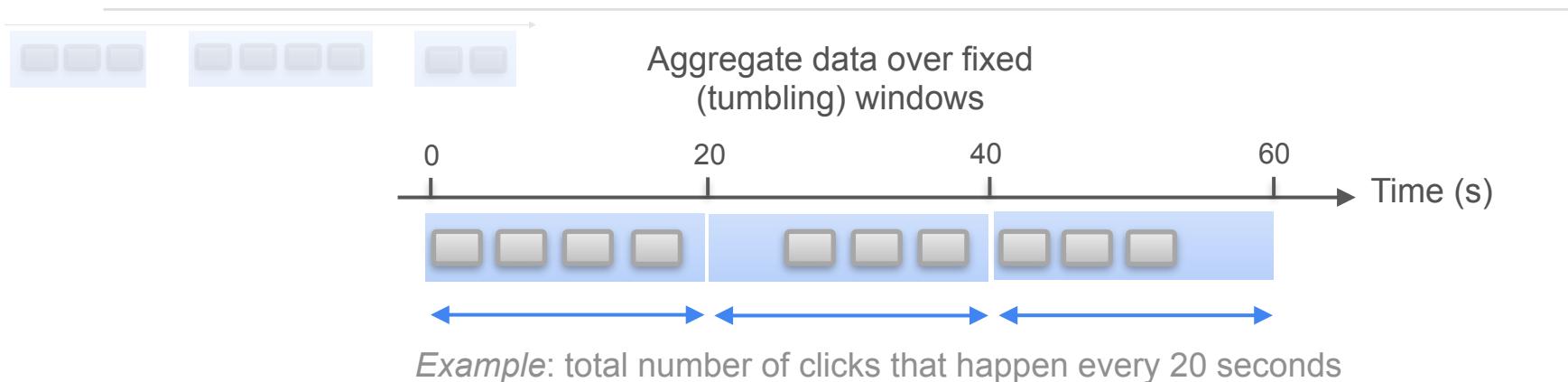


# Queries on Streaming Data

## Windowed Query

Bound your queries using a window & apply operations over that window.  
(e.g. *aggregating, adding, removing* data)

### Fixed-time Window

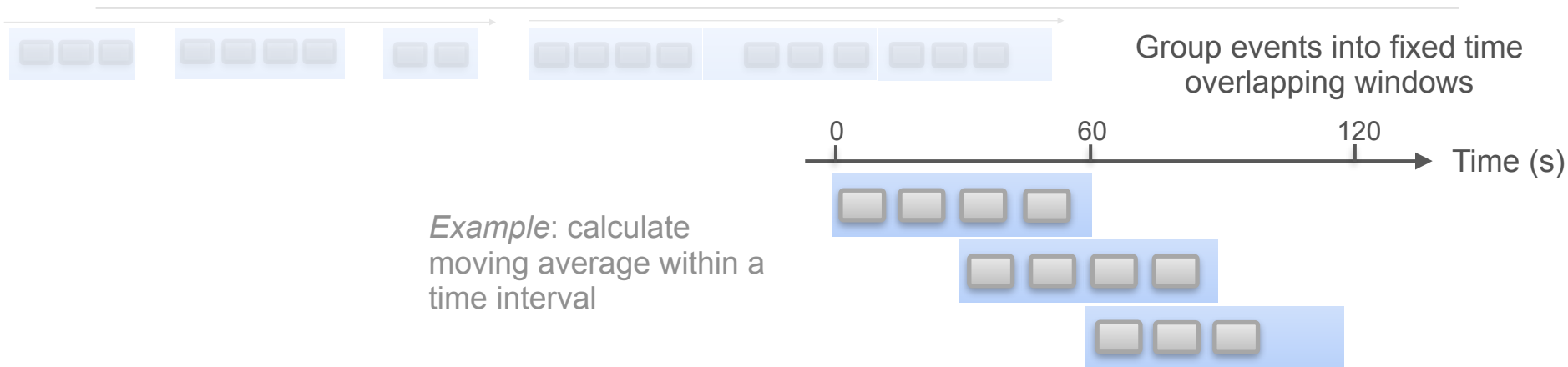


# Queries on Streaming Data

## Windowed Query

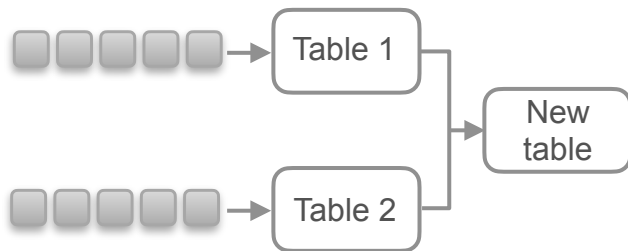
Bound your queries using a window & apply operations over that window.  
(e.g. *aggregating, adding, removing* data)

### Sliding Window

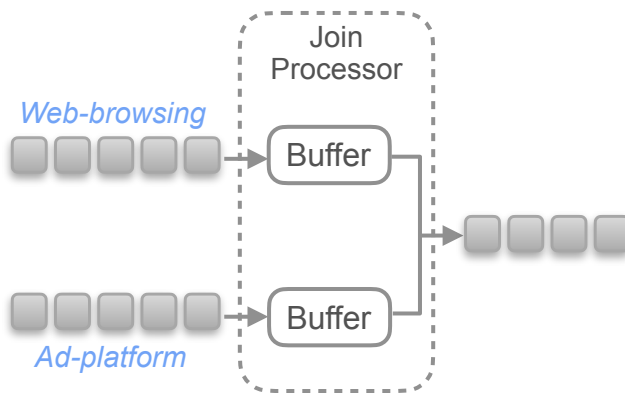


# Joining Data Streams

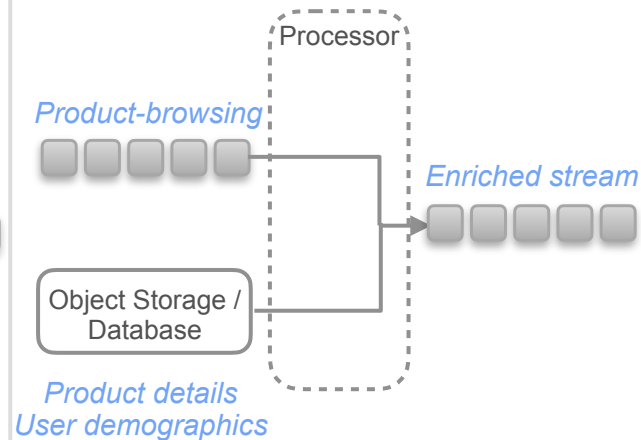
## Conventional Way



## Stream-to-Stream



## Enrich with batch data





DeepLearning.AI

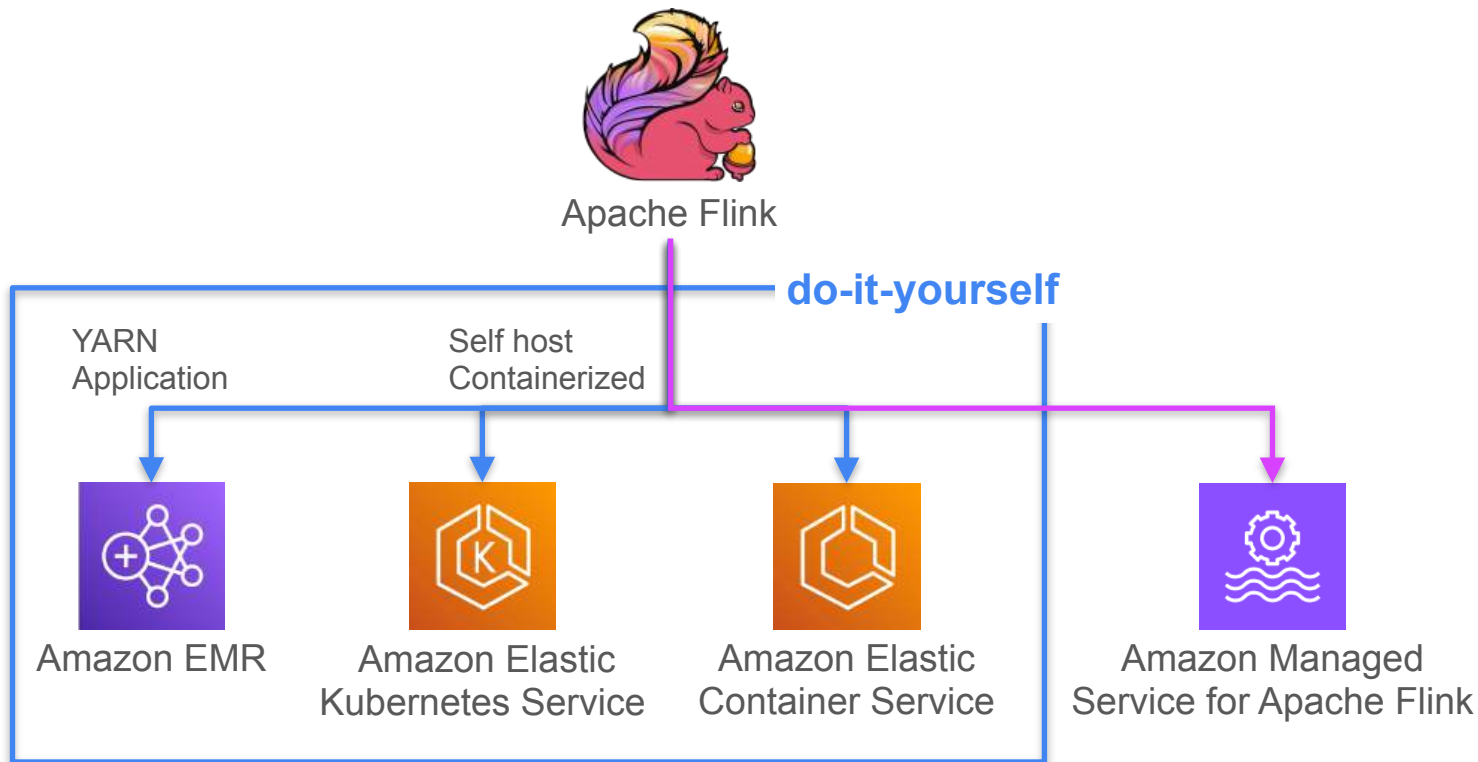
## Queries

---

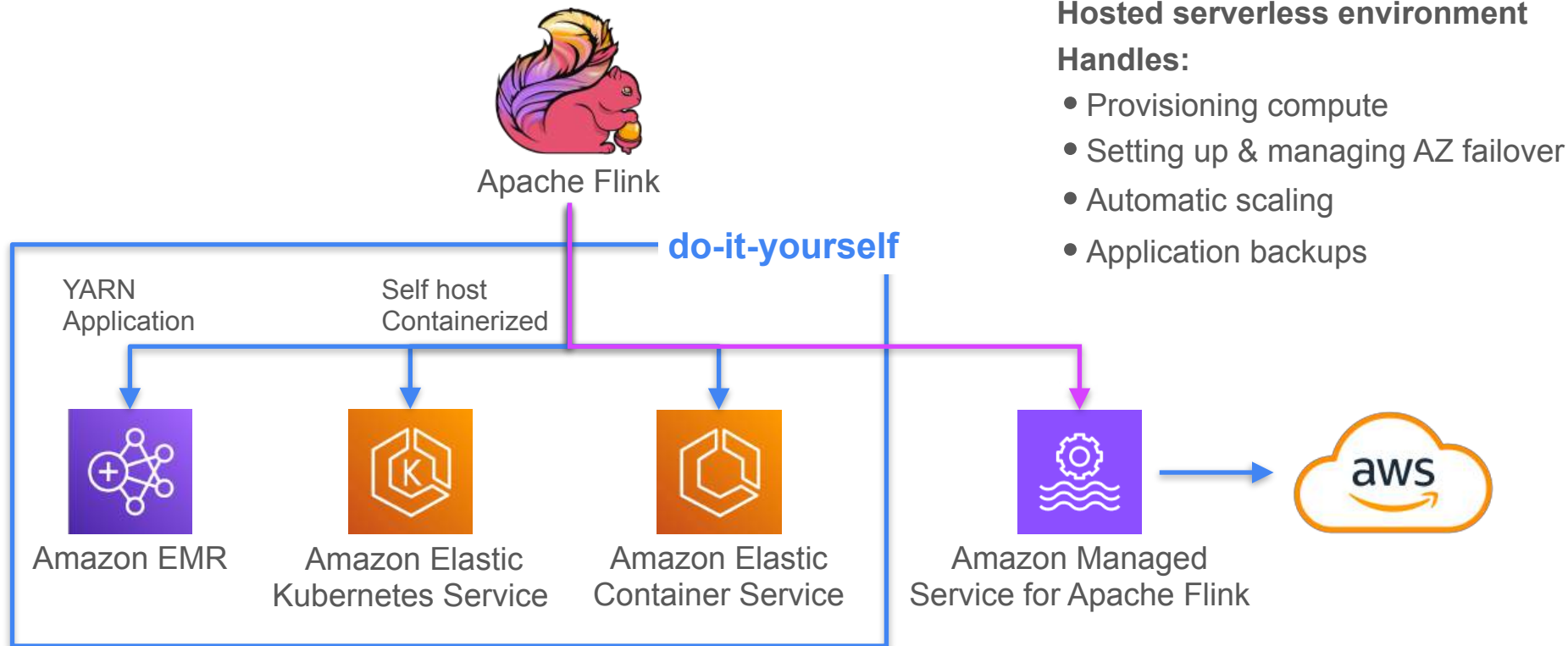
# **Deploying an Application with Amazon Managed Service for Apache Flink**



# Amazon Managed Service for Apache Flink



# Amazon Managed Service for Apache Flink





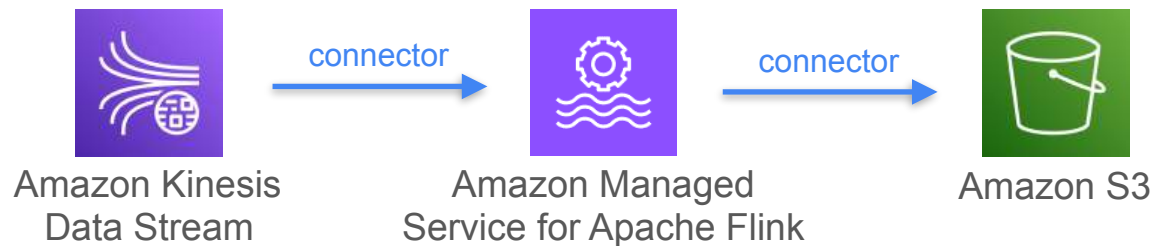
DeepLearning.AI

## Queries

---

# **Deploying a Studio Notebook with Amazon Managed Service for Apache Flink**

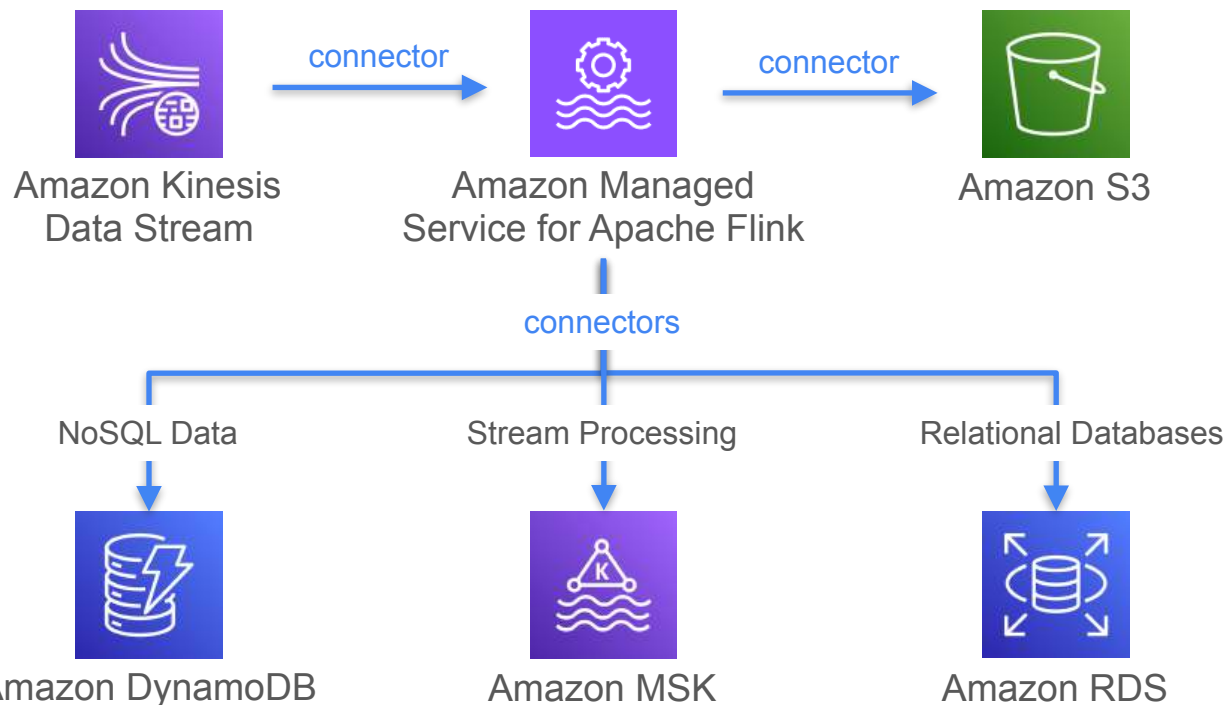
# Amazon Managed Service for Apache Flink



**Connectors provide code for Interfacing with:**

- Databases
- Message queues
- Cloud storage services

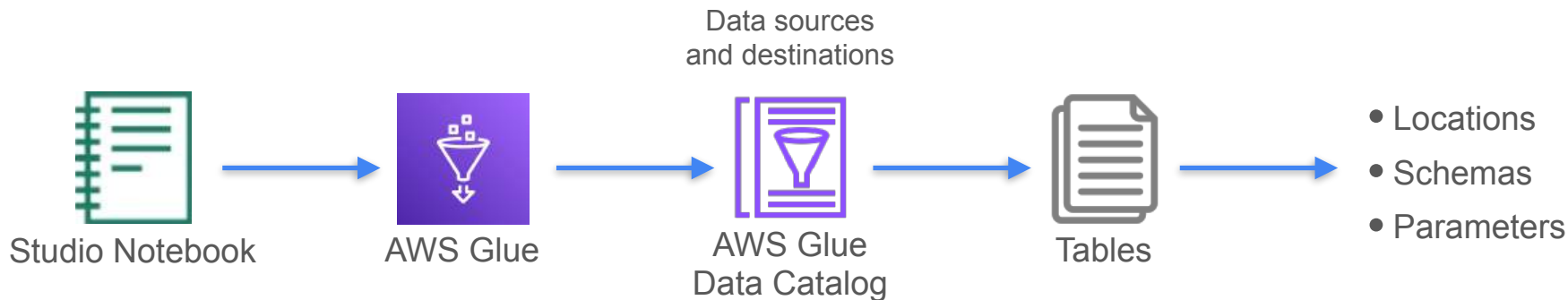
# Amazon Managed Service for Apache Flink



**Connectors provide code for Interfacing with:**

- Databases
- Message queues
- Cloud storage services

# AWS Glue database





DeepLearning.AI

# Queries

---

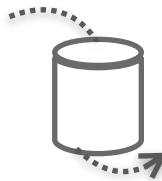
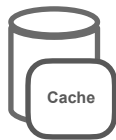
## **Course 3 Summary**

# Week 1

## Storage Abstractions

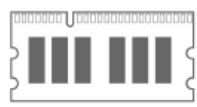


## Storage Systems



## Raw Ingredients

### Physical components



### Processes

Networking

Serialization

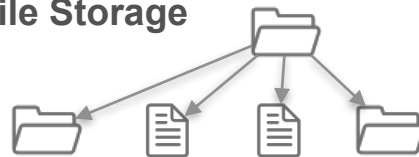
CPU

Compression

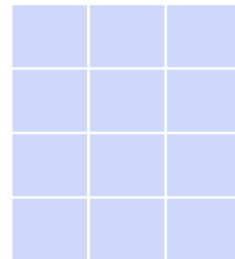
Caching

## Labs

### File Storage



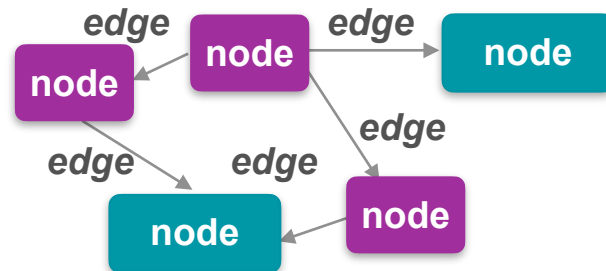
### Block Storage



### Object Storage



## Graph Databases



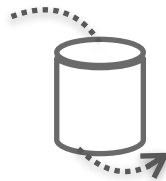
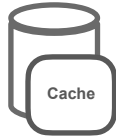


# Week 2

## Storage Abstractions

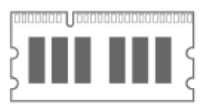


## Storage Systems



## Raw Ingredients

### Physical components



### Processes

Networking

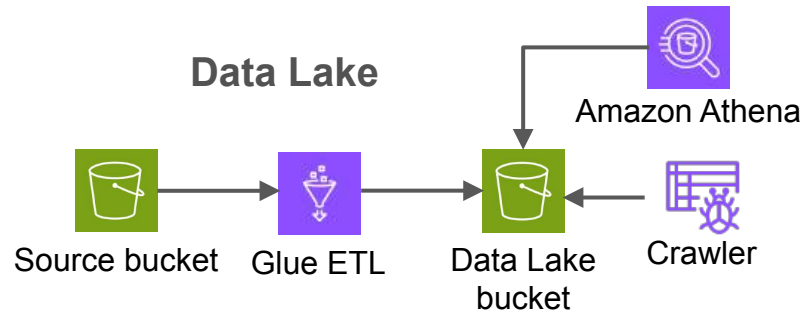
Serialization

CPU

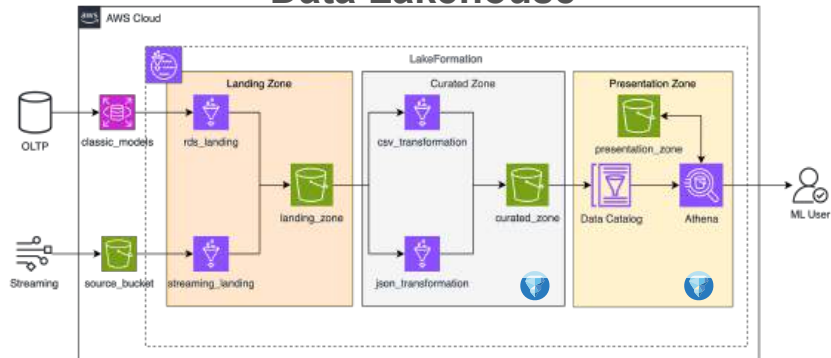
Compression

Caching

## Labs

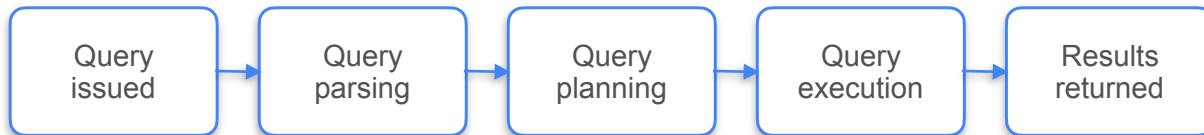


## Data Lakehouse



# Week 3

- Life of a query



- Processing of filtering, joining and aggregating queries

## Labs

- Advanced SQL statements
- Execution time of an analytical query on a row versus columnar storage
- Time-based windowed query on streaming data



Amazon Managed Service  
for Apache Flink