中国数据库暑期学校

Cloud-Native Database Systems

01: The Fundamentals + Snowflake

Huanchen Zhang

清華大學 Tsinghua University

交叉信息研究院 Institute for Interdisciplinary Information Sciences

About Me

Undergrad

PhD

Gap Year

AP

Course Agenda

Day 1: Introduction, Cloud Services,
Storage and Compute Disaggregation
Cloud Data Warehouses I

Day 2: Cloud Data Warehouses I I
Serverless
Cost Intelligence

Course Agenda

Day 4: Paxos/Raft, Two-Phase Commit
Cloud-Native OLTP Systems I

Day 5: Cloud-Native OLTP Systems I I
Advanced Research Topics

Lectures

- 9:00am - 12:00pm
- Lectures are interactive. Interrupt me anytime

Lectures

- 9:00am - 12:00pm
- Lectures are interactive. Interrupt me anytime
- It’s OK to …
  - eat, drink

Lectures

- 9:00am - 12:00pm
- Lectures are interactive. Interrupt me anytime
- It’s OK to …
  - eat, drink
  - sleep

Lectures

- 9:00am - 12:00pm
- Lectures are interactive. Interrupt me anytime
- It’s OK to …
  - eat, drink
  - sleep
  - talk to your neighbors

Lectures

- 9:00am - 12:00pm
- Lectures are interactive. Interrupt me anytime
- It’s OK to …
  - eat, drink
  - sleep
  - talk to your neighbors
  - leave early

Lectures

- ➜ 9:00am - 12:00pm
- ➜ Lectures are interactive. Interrupt me anytime
- ➜ It’s OK to …
  - eat, drink
  - sleep
  - talk to your neighbors
  - leave early

  But please respect each other

Lectures

- 9:00am - 12:00pm
- Lectures are interactive. Interrupt me anytime
- It’s OK to …
  - eat, drink
  - sleep
  - talk to your neighbors
  - leave early

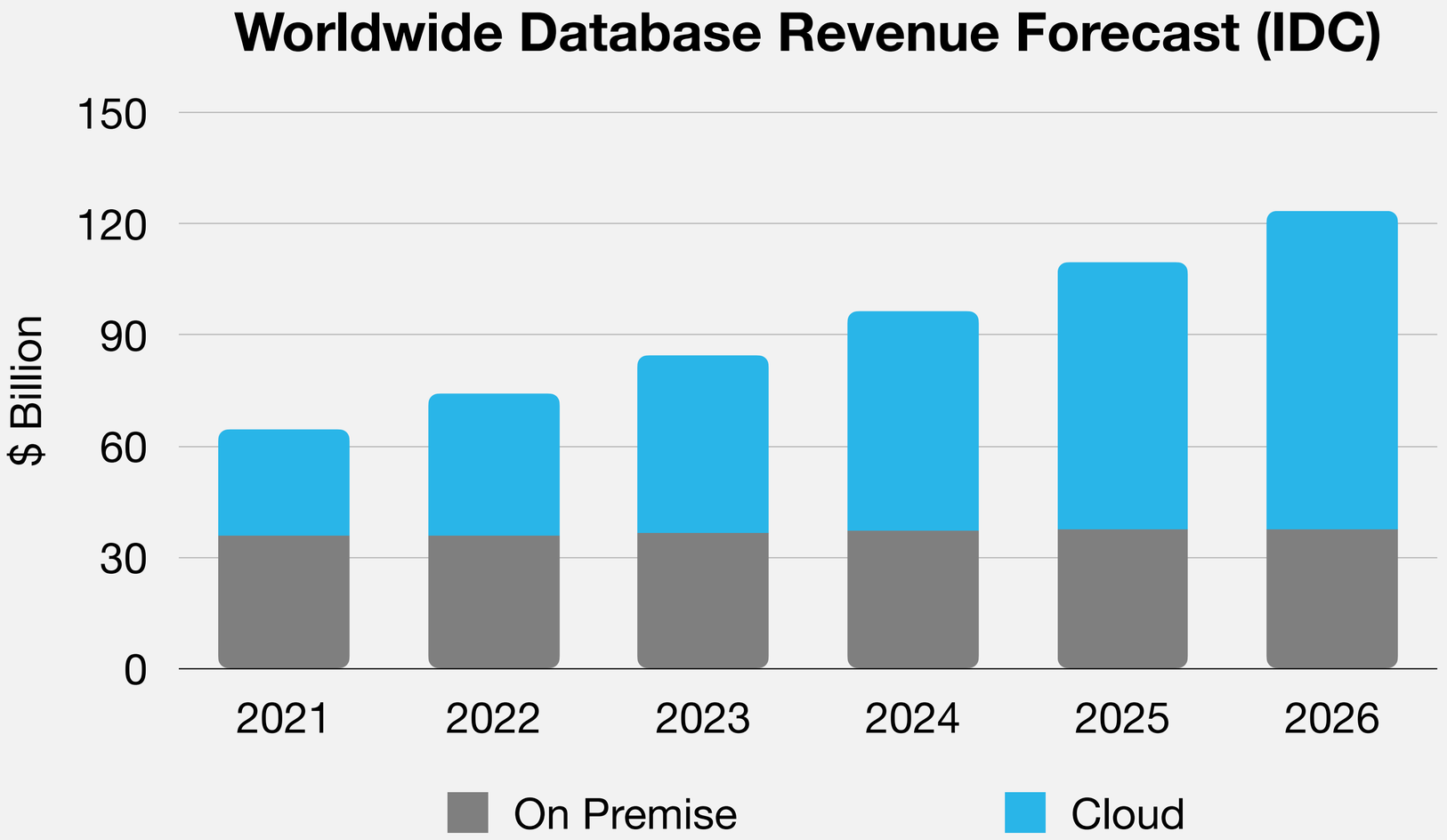
But please respect each other

Lectures

- 9:00am - 12:00pm
- Lectures are interactive. Interrupt me anytime
- It’s OK to …
  - eat, drink
  - sleep
  - talk to your neighbors
  - leave early

But please respect each other

The Booming Cloud Database Market



Why Are Cloud Databases Different?

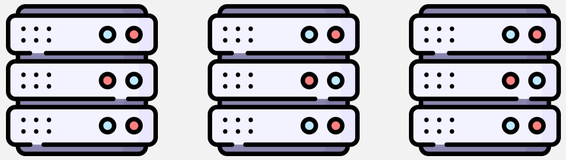
Traditional

```
0110
1001
1010
```

Why Are Cloud Databases Different?

Traditional

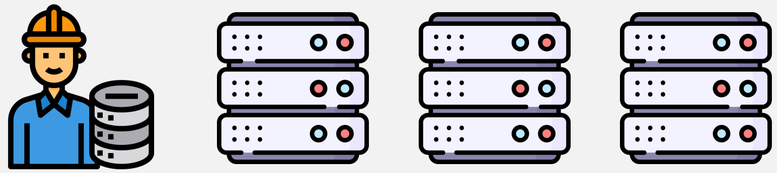
```
0110
1001
1010
```



Why Are Cloud Databases Different?

Traditional

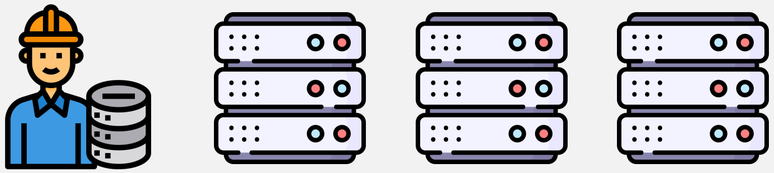
```
0110
1001
1010
```



Why Are Cloud Databases Different?

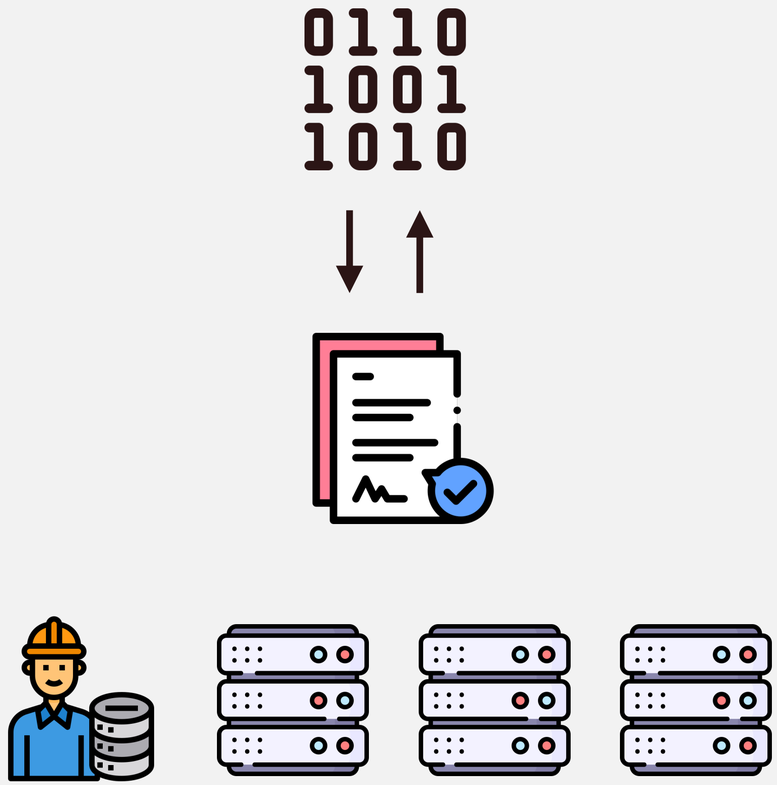
Traditional

```
0110
1001
1010
```



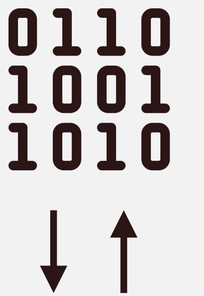
Why Are Cloud Databases Different?

Traditional

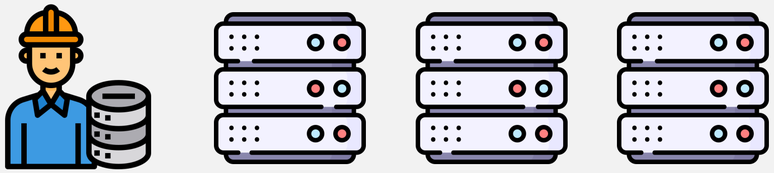


Why Are Cloud Databases Different?

Traditional



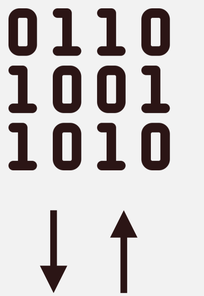
$$$



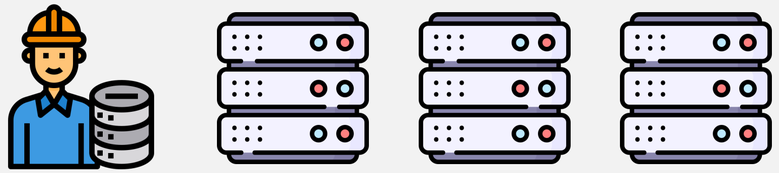
$$$

Why Are Cloud Databases Different?

Traditional

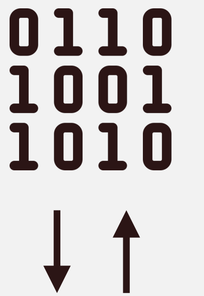


$$$

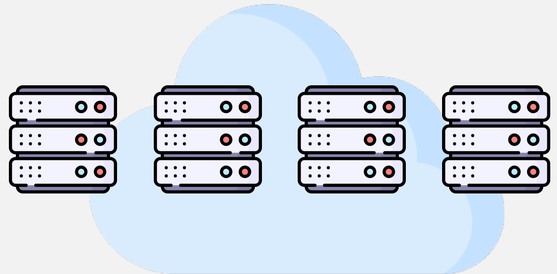


$$$

Cloud-Native



$ pay-as-you-go



• • • Elasticity

Why Are Cloud Databases Different?

Traditional

Cloud-Native

“云原生带来的最大技术红利以及经济红利就是规模化应用后带来边际成本下降效应”

Traditional vs. Cloud Data Warehouses

	Traditional DW	Cloud DW
Capital Cost		
Operational Cost		
Elasticity		
Availability		
Software Upgrade		
Performance		
Security		

Traditional vs. Cloud Data Warehouses

	Traditional DW	Cloud DW
Capital Cost	High	None
Operational Cost		
Elasticity		
Availability		
Software Upgrade		
Performance		
Security		

Traditional vs. Cloud Data Warehouses

	Traditional DW	Cloud DW
Capital Cost	High	None
Operational Cost	High	Low
Elasticity		
Availability		
Software Upgrade		
Performance		
Security		

Traditional vs. Cloud Data Warehouses

	Traditional DW	Cloud DW
Capital Cost	High	None
Operational Cost	High	Low
Elasticity	Poor	Good
Availability		
Software Upgrade		
Performance		
Security		

Traditional vs. Cloud Data Warehouses

	Traditional DW	Cloud DW
Capital Cost	High	None
Operational Cost	High	Low
Elasticity	Poor	Good
Availability	Medium	Good
Software Upgrade		
Performance		
Security		

Traditional vs. Cloud Data Warehouses

	Traditional DW	Cloud DW
Capital Cost	High	None
Operational Cost	High	Low
Elasticity	Poor	Good
Availability	Medium	Good
Software Upgrade	Slow	Fast
Performance		
Security		

Traditional vs. Cloud Data Warehouses

	Traditional DW	Cloud DW
Capital Cost	High	None
Operational Cost	High	Low
Elasticity	Poor	Good
Availability	Medium	Good
Software Upgrade	Slow	Fast
Performance	Could be Better	Medium
Security		

Traditional vs. Cloud Data Warehouses

	Traditional DW	Cloud DW
Capital Cost	High	None
Operational Cost	High	Low
Elasticity	Poor	Good
Availability	Medium	Good
Software Upgrade	Slow	Fast
Performance	Could be Better	Medium
Security	Medium	Medium +

Today

- ➜ Cloud Computing & Cloud Services
- ➜ Storage and Compute Disaggregation
- ➜ Cloud Data Warehouses

Today

- **Cloud Computing & Cloud Services**
- Storage and Compute Disaggregation
- Cloud Data Warehouses

Cloud Computing

➜ Delivery of computing as a **service** over the network

- Backed by a large distributed computing infrastructure

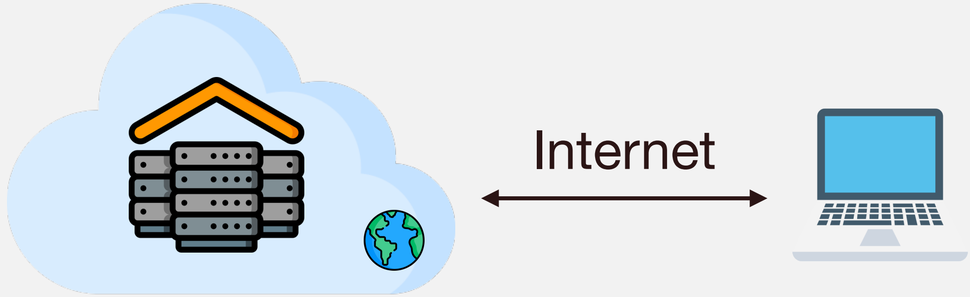
➜ Cloud computing as **utility**

- Pay as you go
- Cloud providers provision resources rapidly

Types of Clouds

➔ **Public Cloud**

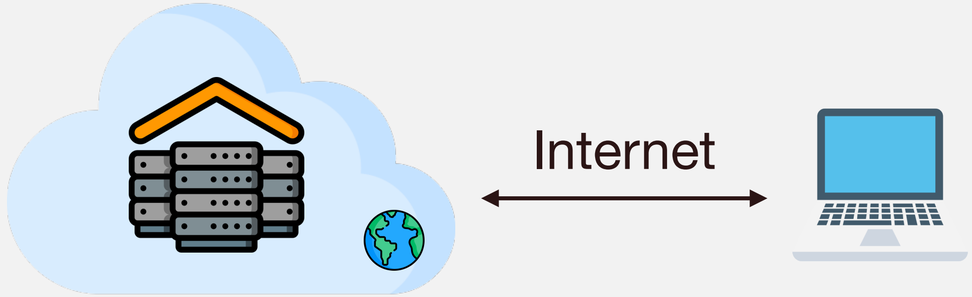
- Owned by a cloud provider, made available to public via internet



Types of Clouds

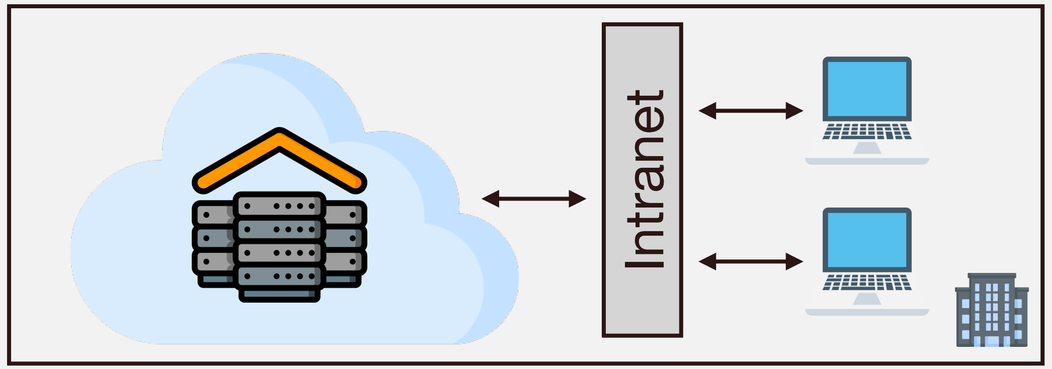
➔ **Public Cloud**

- Owned by a cloud provider, made available to public via internet



➔ **Private Cloud**

- Owned and accessed only by organization



Types of Clouds

➜ **Public Cloud**

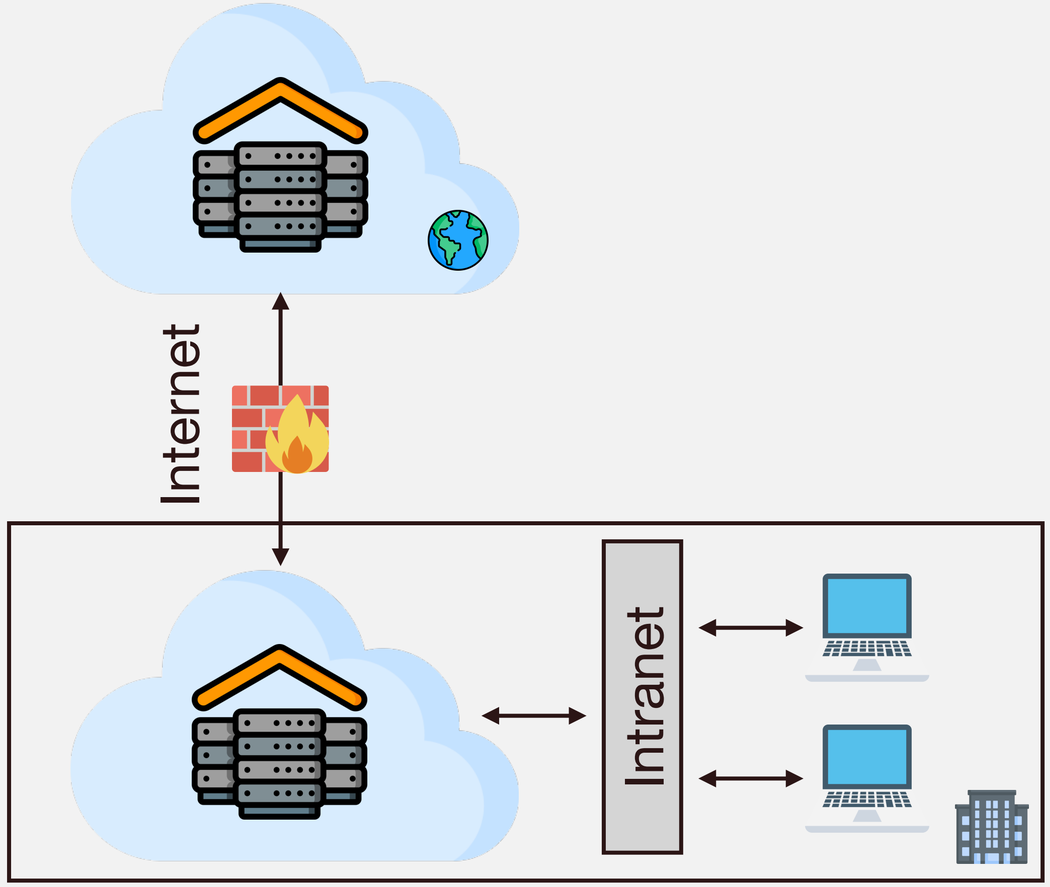
- Owned by a cloud provider, made available to public via internet

➜ **Private Cloud**

- Owned and accessed only by organization

➜ **Hybrid Cloud**

- Private cloud + elastic public cloud (to handle burst of requests)



Types of Clouds

➜ **Public Cloud**

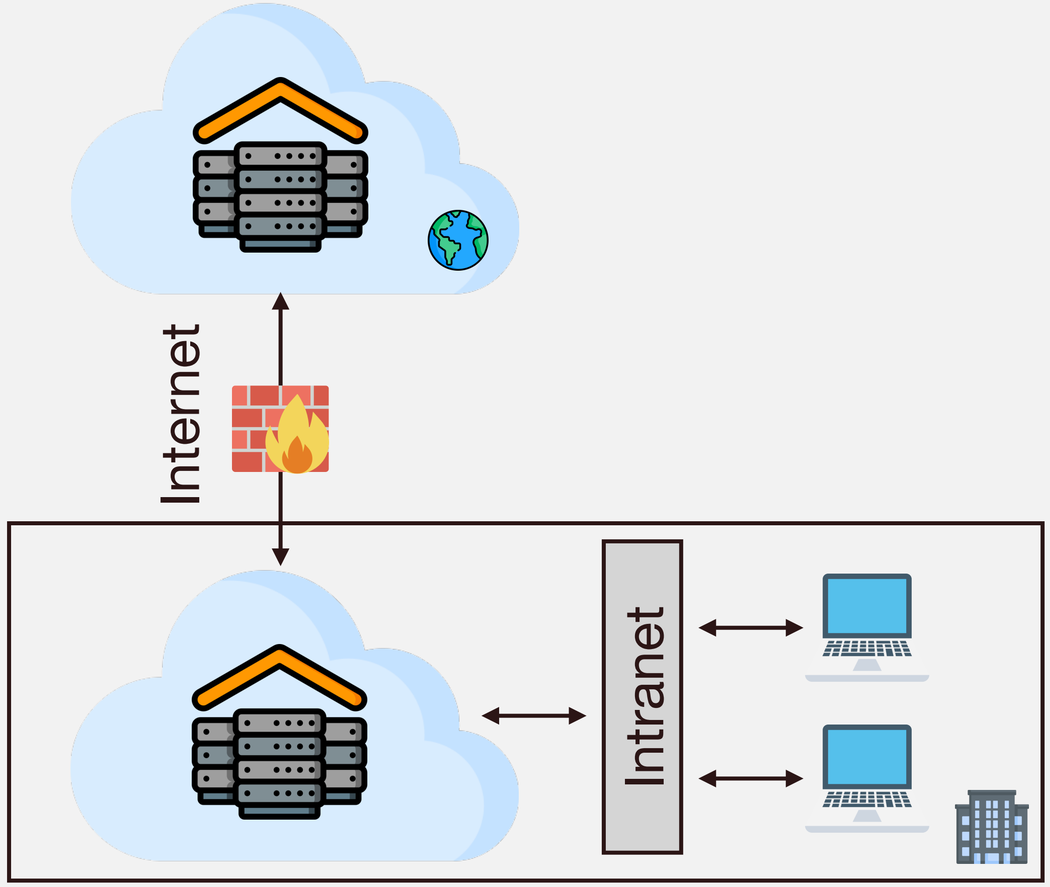
- Owned by a cloud provider, made available to public via internet

➜ **Private Cloud**

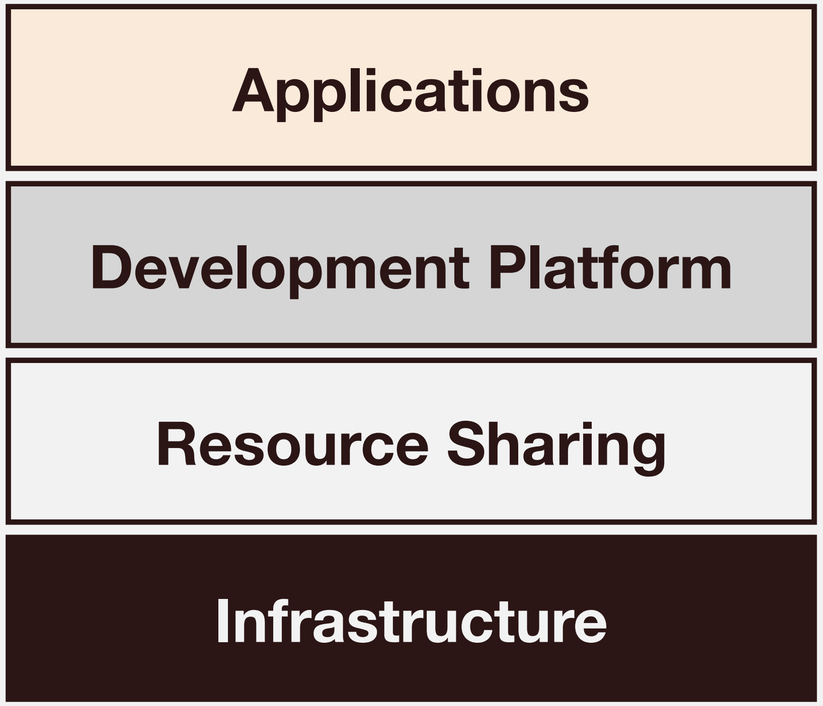
- Owned and accessed only by organization

➜ **Hybrid Cloud**

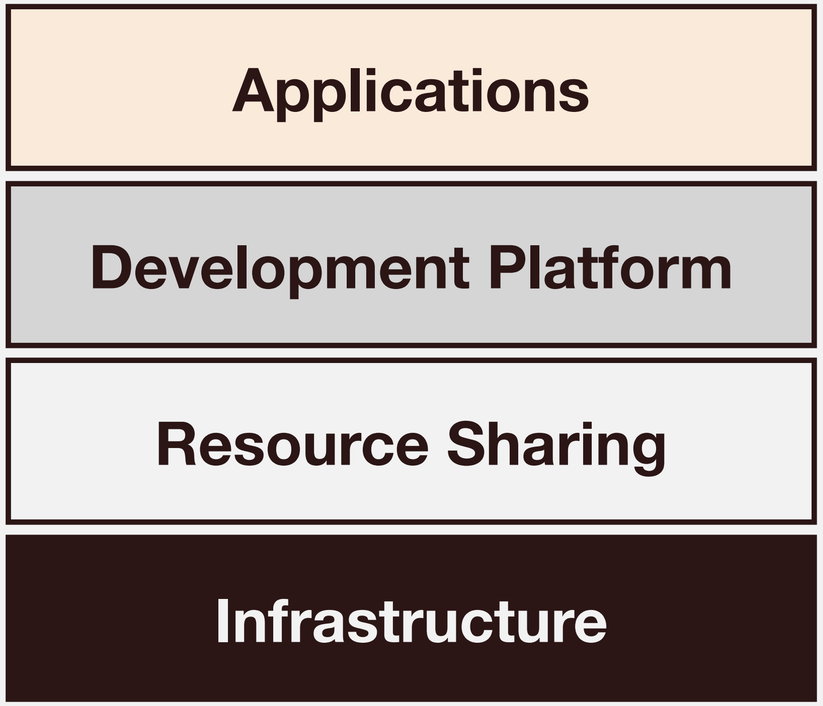
- Private cloud + elastic public cloud (to handle burst of requests)



Cloud Building Blocks

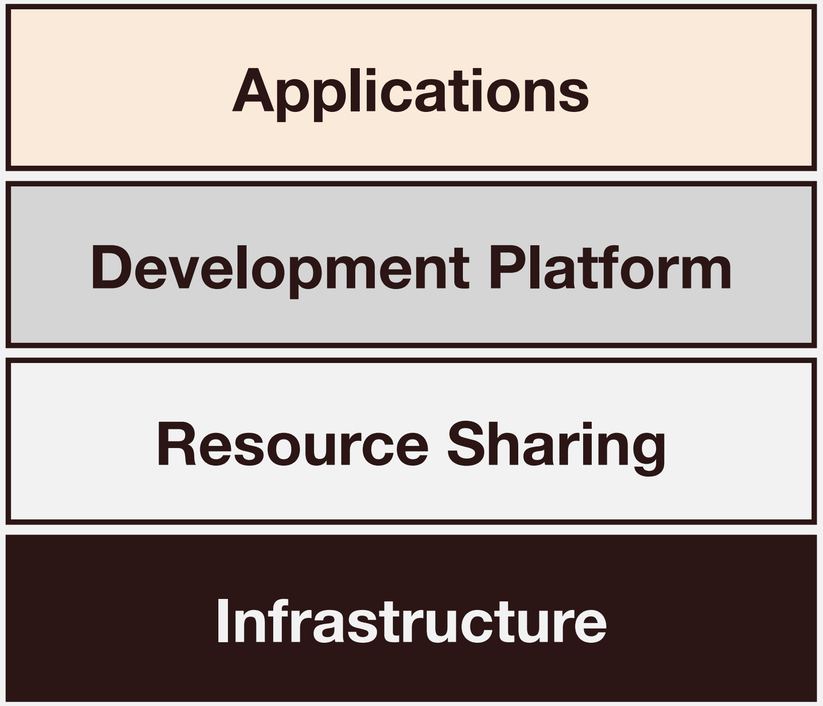


Cloud Building Blocks



Physical Servers

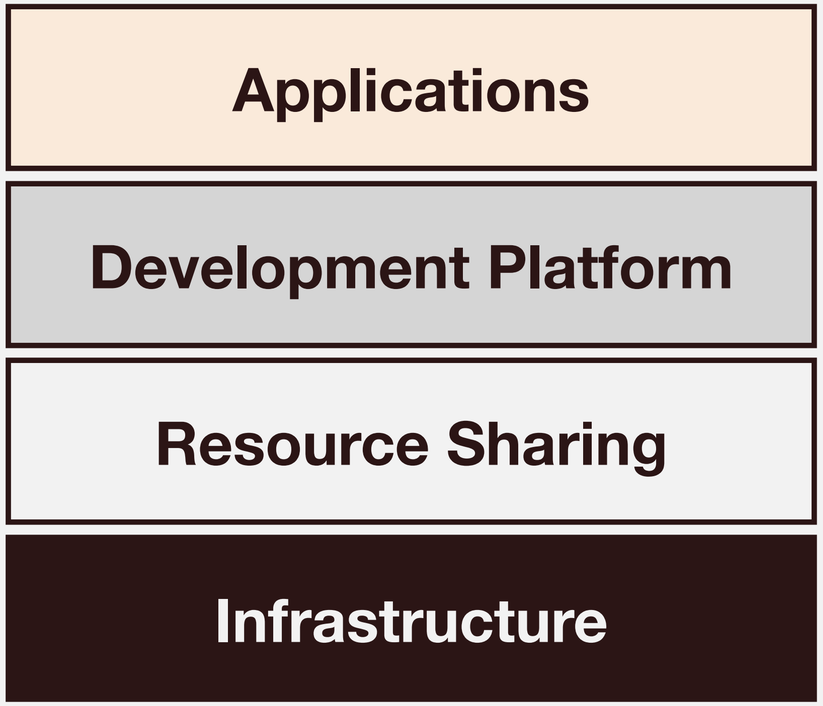
Cloud Building Blocks



Virtualizing Resources

Physical Servers

Cloud Building Blocks

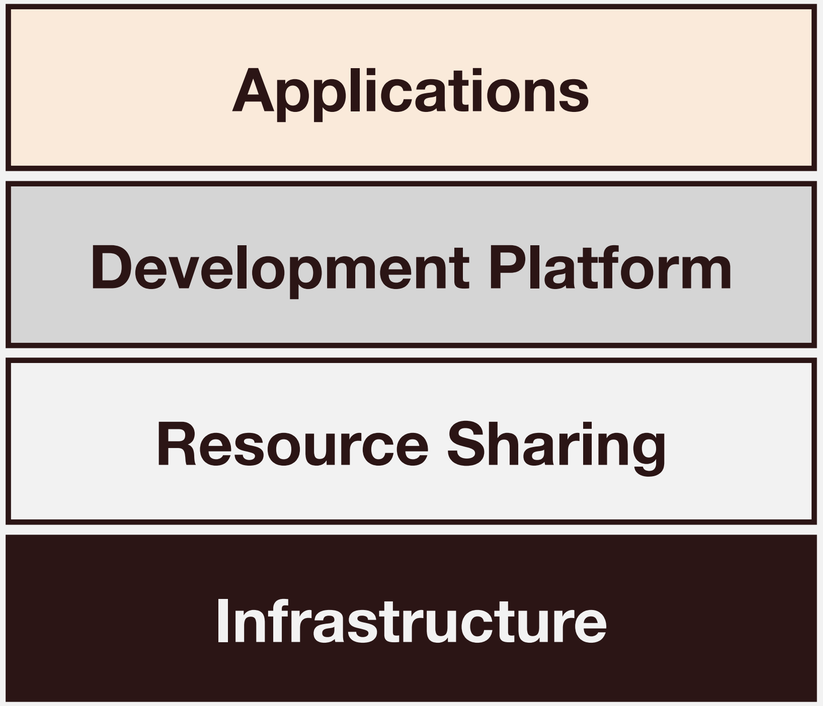


E.g., Google App Engine, AWS Lambda

Virtualizing Resources

Physical Servers

Cloud Building Blocks



E.g., Email, Google Drive

E.g., Google App Engine, AWS Lambda

Virtualizing Resources

Physical Servers

Cloud Computing Services

➜ Infrastructure as a Service (**IaaS**)

- Leasing (virtualized) infrastructure remotely
- Configurable CPU, memory, disk, and network bandwidth
- Resource sharing, sandboxing
- Most flexibility in software development

Amazon EC2

Amazon S3

Cloud Computing Services

- ➜ Infrastructure as a Service (**IaaS**)
  - Leasing (virtualized) infrastructure remotely
  - Configurable CPU, memory, disk, and network bandwidth
  - Resource sharing, sandboxing
  - Most flexibility in software development

Amazon EC2

Amazon S3

- ➜ Platform as a Service (**PaaS**)
  - To facilitate creation of cloud software
  - Abstract away the management of underlying infrastructure
  - Built-in scaling

AWS Lambda

App Engine

Cloud Computing Services

➜ Software as a Service (**SaaS**)

- Software and data are hosted on the cloud
- Often accessed by using a web browser or an mobile app
- Multitenant
- Provider handles software updates and patches

Cloud Computing Services

- ➜ Software as a Service (**SaaS**)
  - Software and data are hosted on the cloud
  - Often accessed by using a web browser or an mobile app
  - Multitenant
  - Provider handles software updates and patches
  - **Limitations**
    - Vendor lock-in
    - Hard to customize
    - Data security

Common AWS Services

Storage:

S3

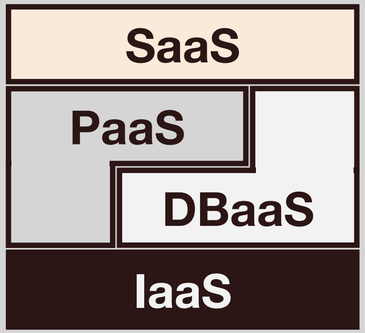
EBS

Compute:

EC2

Lambda

Database:



RDS
Aurora

Amazon **Redshift**

amazon
DynamoDB

Big Data:

Athena

amazon
EMR

Amazon
SageMaker

• • •

Simple Storage Service (S3)

S3

➜ **Object Storage**

- Just want to store some bytes

➜ **Key-Value API**

- Each object is associated with a key (unique with a bucket)
- **PUT**(key, value): create/replace a whole object
- **GET**(key, <range>): read a byte range in the object
- **LIST**(<startKey>): list keys in the bucket in order (return 1000 per call)

Simple Storage Service (S3)

S3

- **Object Storage**
  - Just want to store some bytes
- **Key-Value API**
  - Each object is associated with a key (unique with a bucket)
  - **PUT**(key, value): create/replace a whole object
  - **GET**(key, <range>): read a byte range in the object
  - **LIST**(<startKey>): list keys in the bucket in order (return 1000 per call)
- **Eventual Consistency**
  - But guarantees read-your-own-writes

Simple Storage Service (S3)

Simple key-value HTTP(S) interface

No in-place update, object must be written in full

But can read part of an object

S3

Simple Storage Service (S3)

- ➕ Simple key-value HTTP(S) interface
- ➖ No in-place update, object must be written in full
- ➕ But can read part of an object
- ➖ High access **latency** (10s - 100s ms)
- ➕ Aggregated **bandwidth** scales well
- ➖ Performance could vary

Simple Storage Service (S3)

S3

- \+ Simple key-value HTTP(S) interface
- − No in-place update, object must be written in full
- \+ But can read part of an object

- − High access **latency** (10s - 100s ms)
- \+ Aggregated **bandwidth** scales well
- − Performance could vary

- \+ **Unbeatable** availability and **durability** (11 9's)
- \+ Super **cheap** ($1-23 /TB/month)

Simple Storage Service (S3)

- ➕ Simple key-value HTTP(S) interface
- ➖ No in-place update, object must be written in full
- ➕ But can read part of an object

- ➖ High access **latency** (10s - 100s ms)
- ➕ Aggregated **bandwidth** scales well
- ➖ Performance could vary

- ➕ **Unbeatable** availability and **durability** (11 9's)
- ➕ Super **cheap** ($1-23 /TB/month)

➜ **Ideal Use Case**

- Large Data Volume
- Data mostly immutable
- Latency-insensitive app

Elastic Compute Cloud (EC2)

- ➜ Rent virtual machine instances
- ➜ A wide range of instance types
  - Overall “size”
  - Different optimization emphasis: CPU, memory, storage, GPU, …
- ➜ Pricing
  - On-demand: pay-as-you-go
  - Reserved instance
  - Spot instance

Lambda

➜ Serverless Computing

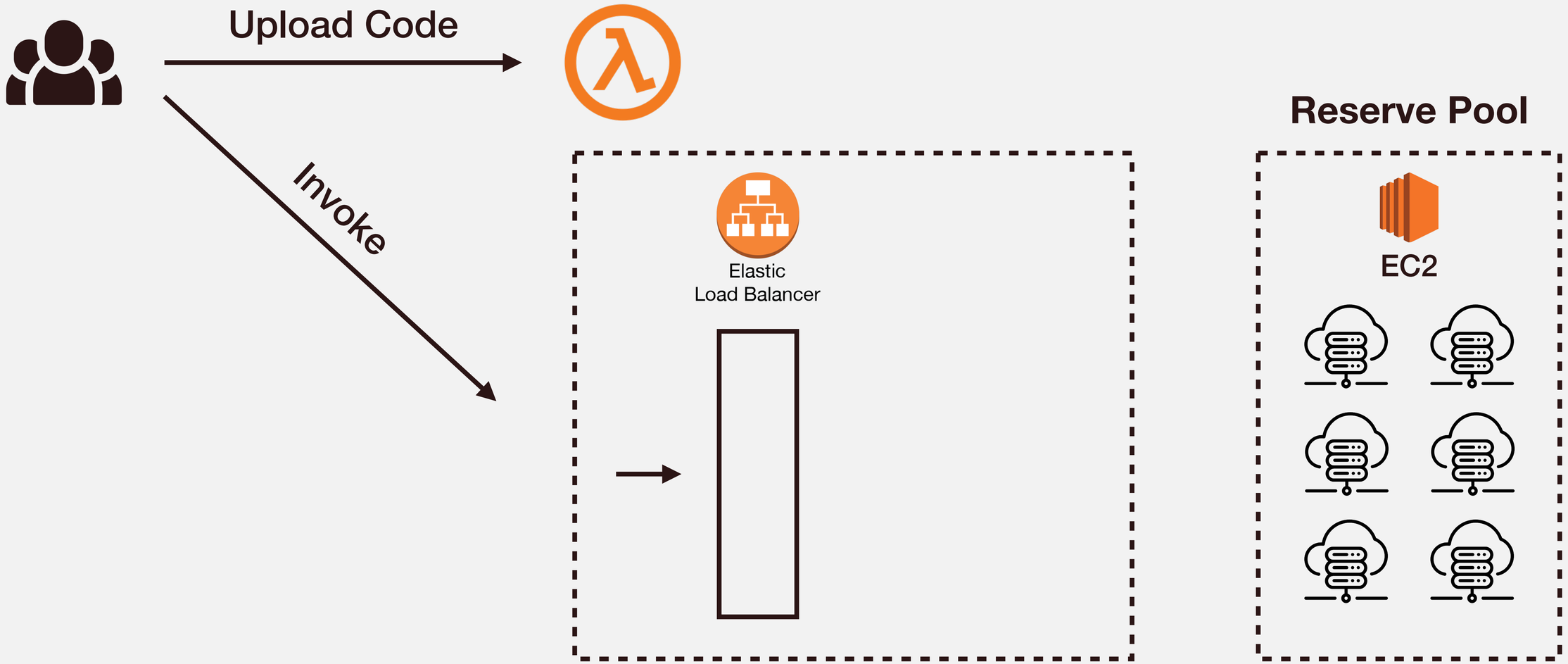
Lambda

"… is a cloud computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of machine resources"
— Wikipedia

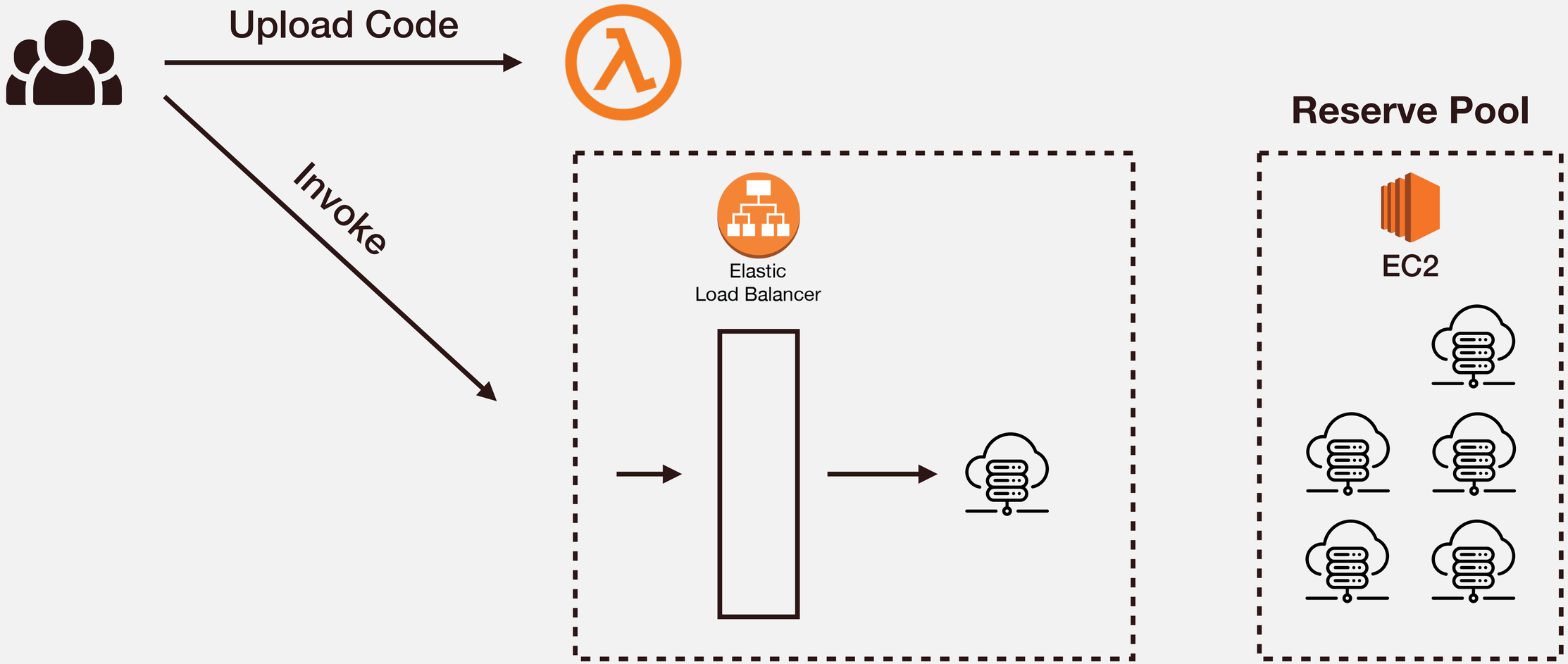
- Cloud providers execute code for developers
- A.k.a. **Function as a Service (FaaS)**
- Developers do not need to worry about:
  - Instance configuration, management, …
  - Resource provisioning & Scaling
  - Fault-tolerance
- No persistent states

Lambda

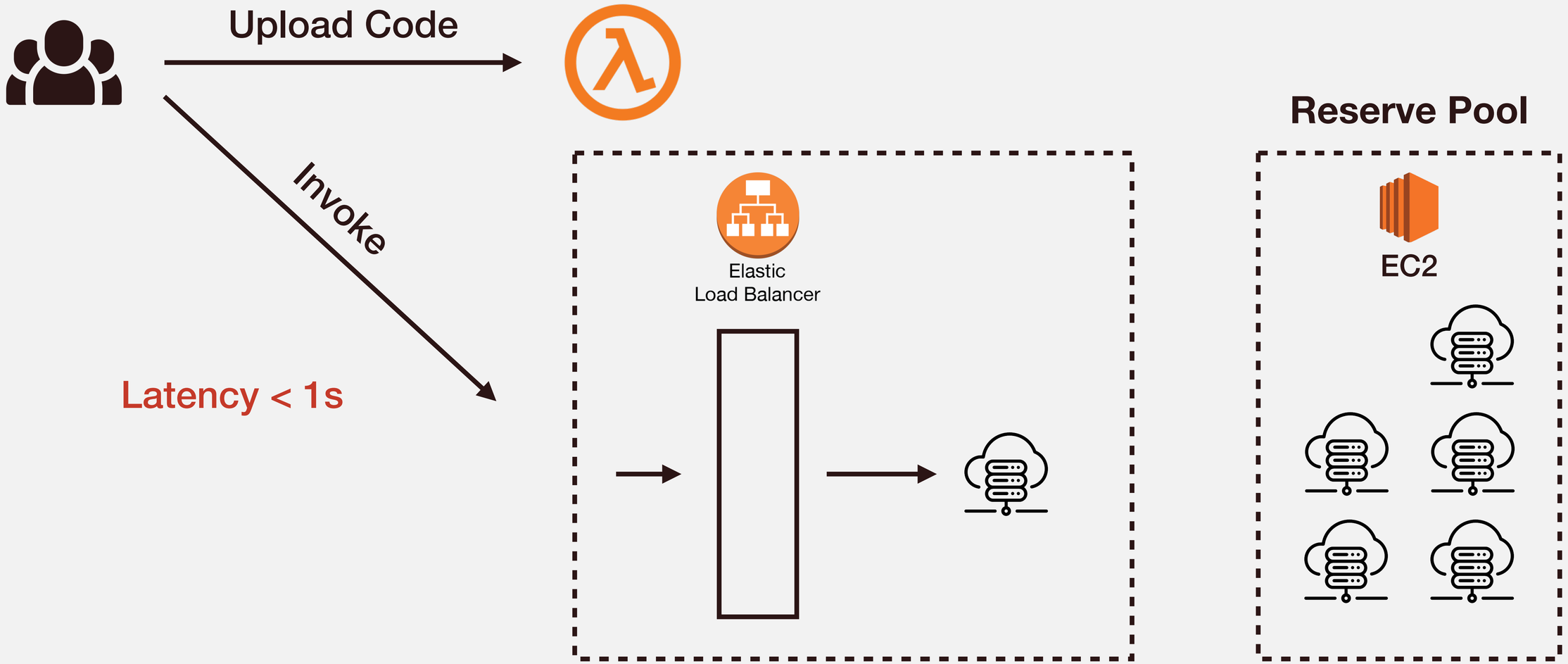
Lambda



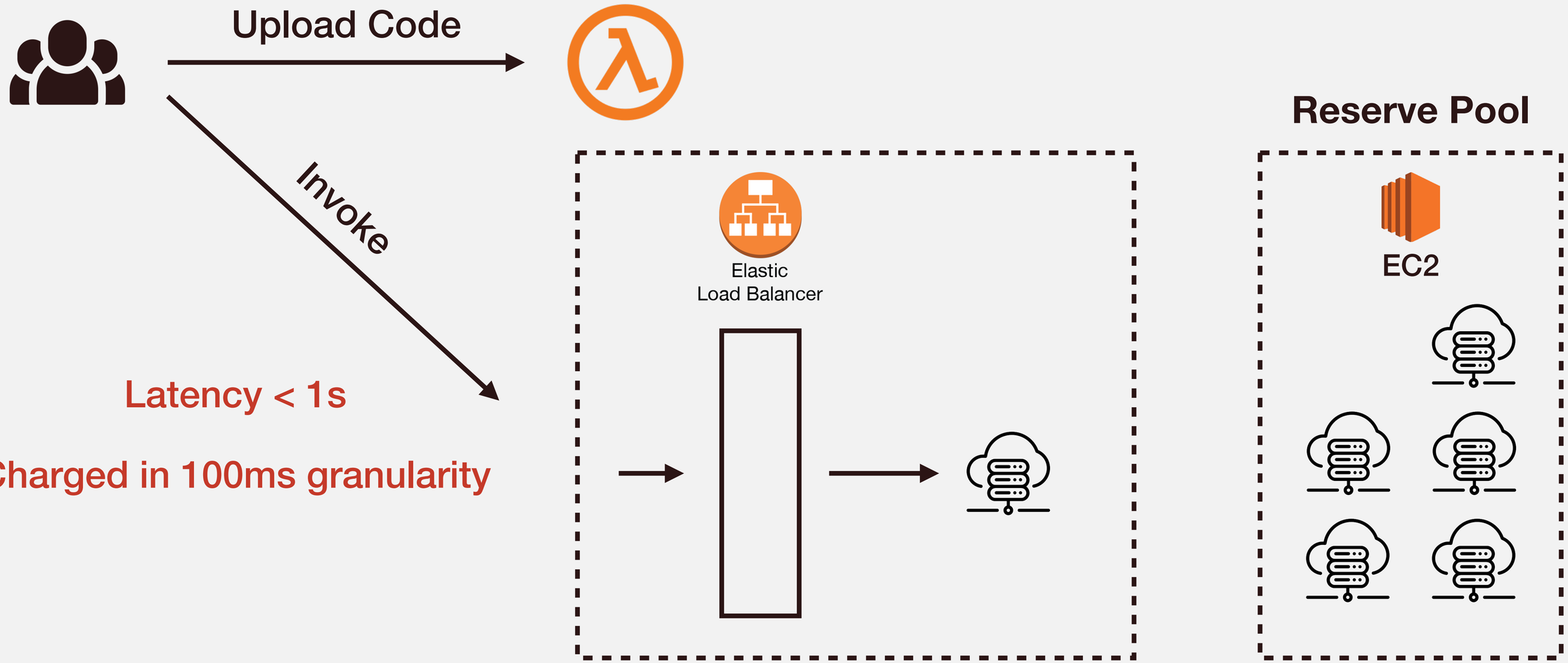
Lambda



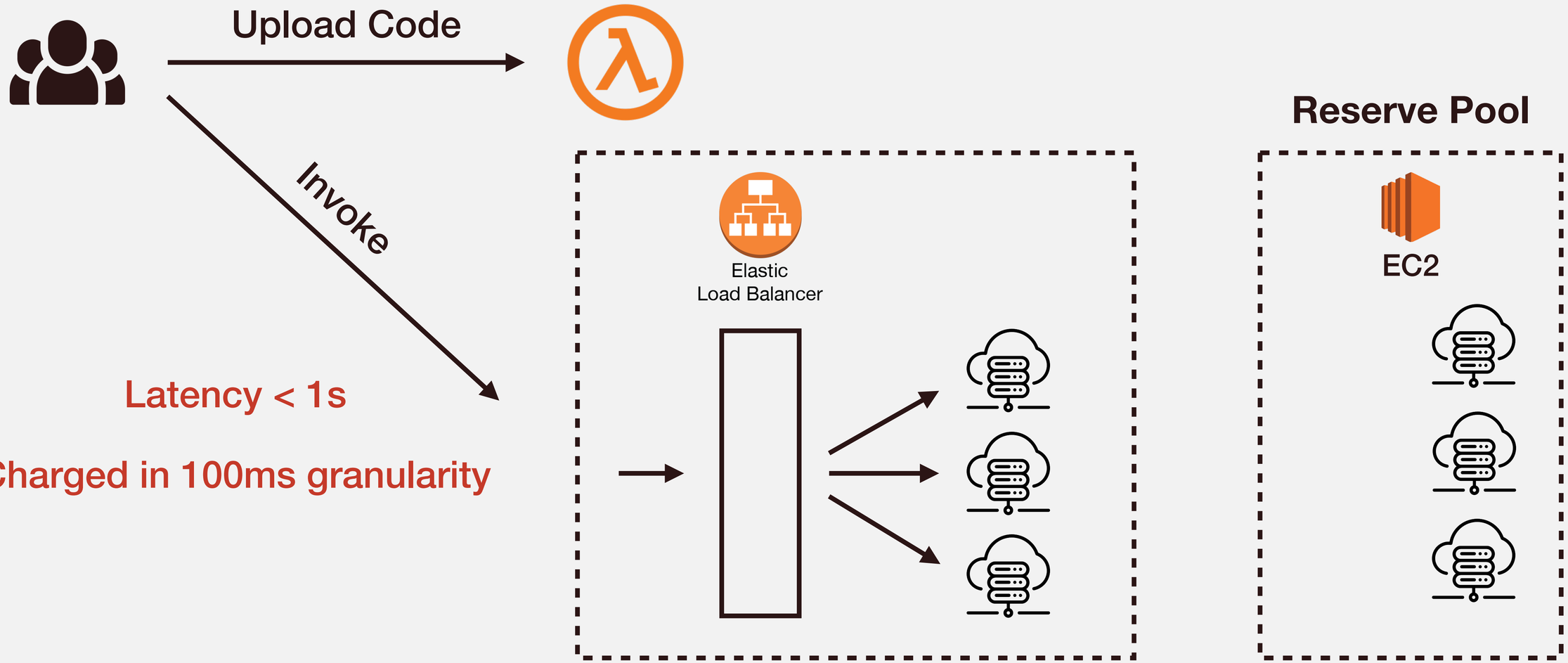
Lambda


Upload Code
Invoke
Latency < 1s
Elastic
Load Balancer
Reserve Pool
EC2

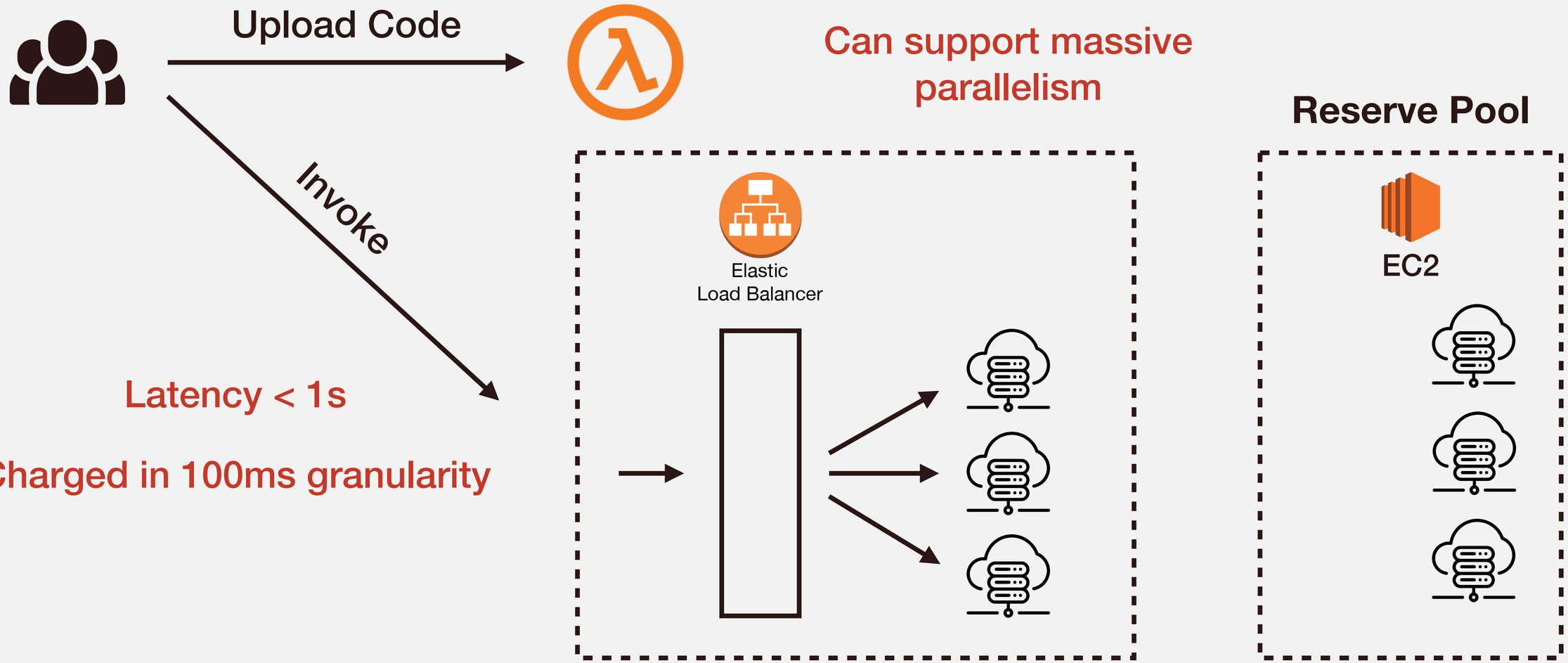
Lambda


Upload Code
Invoke
Latency < 1s
Charged in 100ms granularity
Elastic
Load Balancer
Reserve Pool
EC2

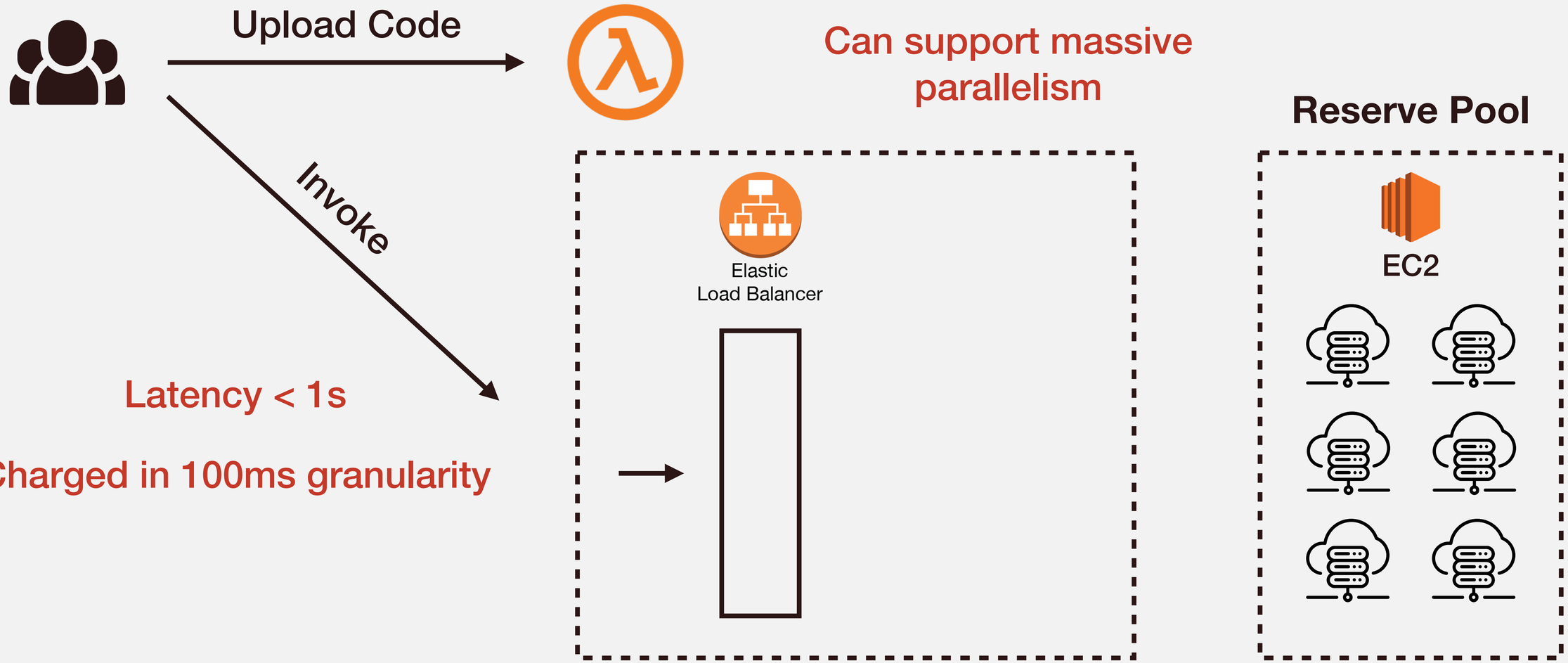
Lambda


Upload Code
Invoke
Latency < 1s
Charged in 100ms granularity
Elastic
Load Balancer
Reserve Pool
EC2

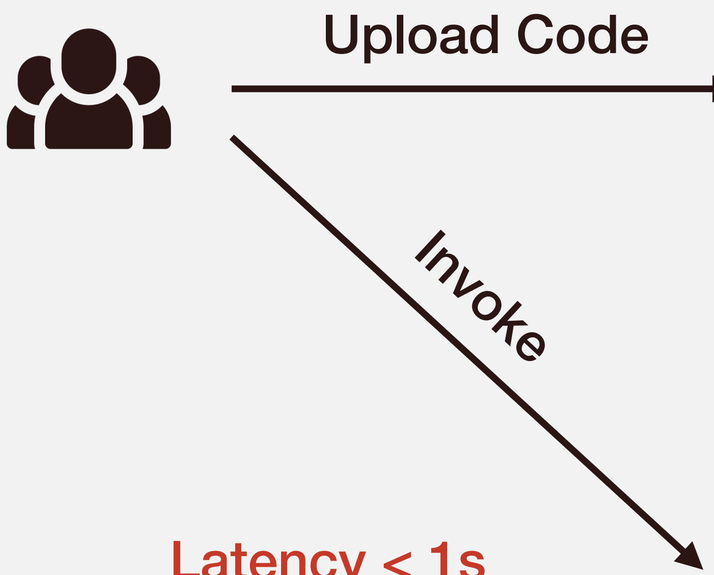
Lambda


Upload Code
Invoke
Latency < 1s
Charged in 100ms granularity
Can support massive parallelism
Elastic
Load Balancer
Reserve Pool
EC2

Lambda


Upload Code
Invoke
Latency < 1s
Charged in 100ms granularity
Can support massive parallelism
Elastic
Load Balancer
Reserve Pool
EC2

Lambda



Latency < 1s

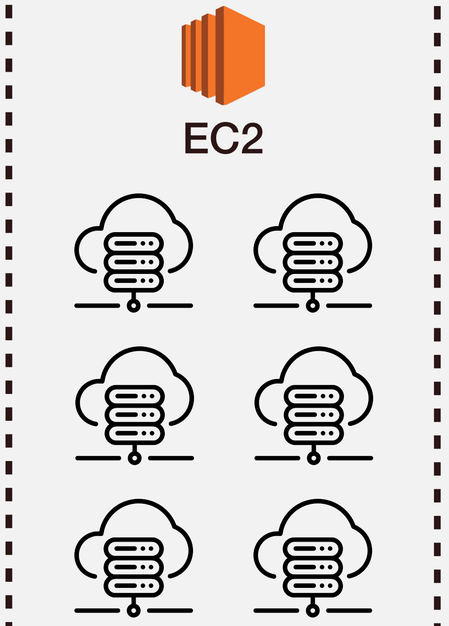
Charged in 100ms granularity

Can support massive parallelism

Limitations

- Limited execution time (15min)
- Limited resource
- No direct communication

Reserve Pool

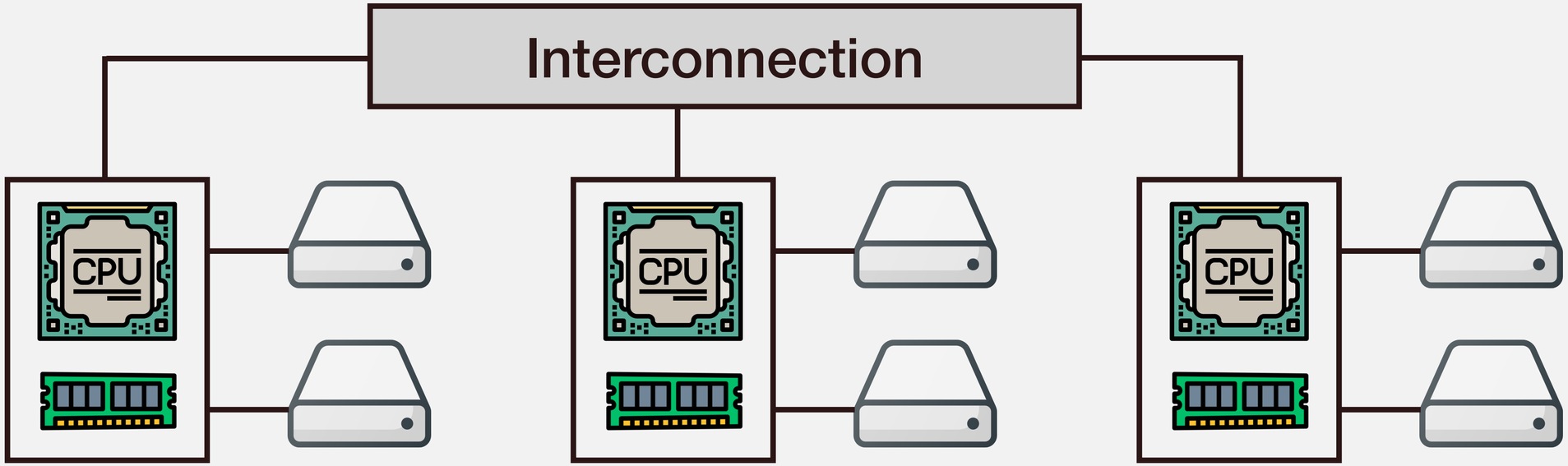


Today

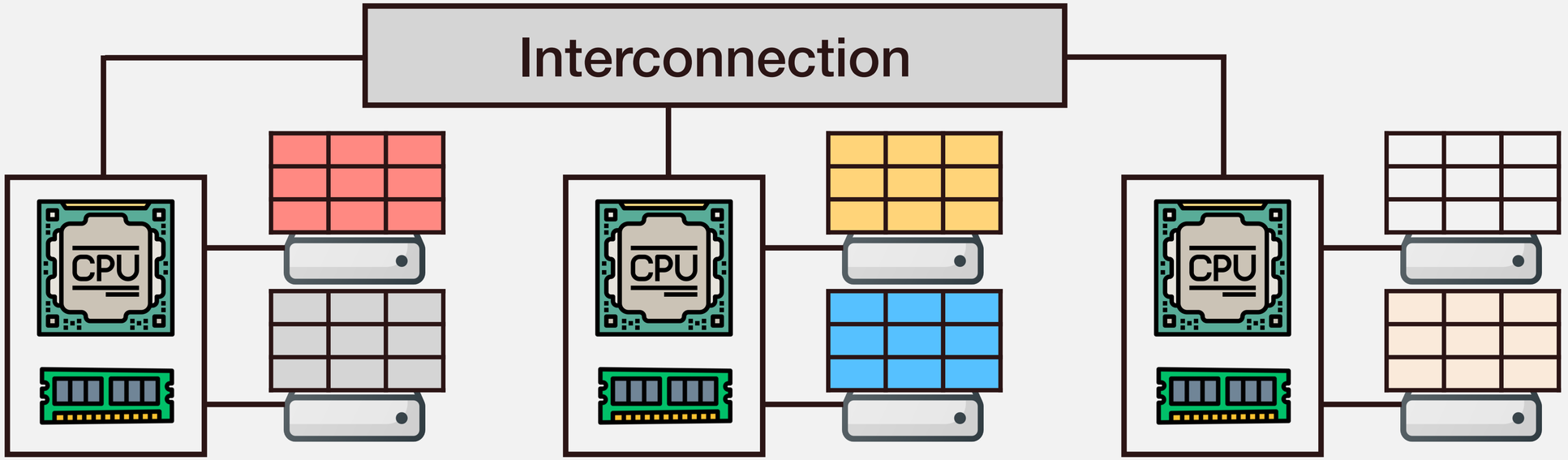
- ➜ Cloud Computing & Cloud Services
- ➜ **Compute and Storage Disaggregation**
- ➜ Cloud Data Warehouses

Compute  Storage

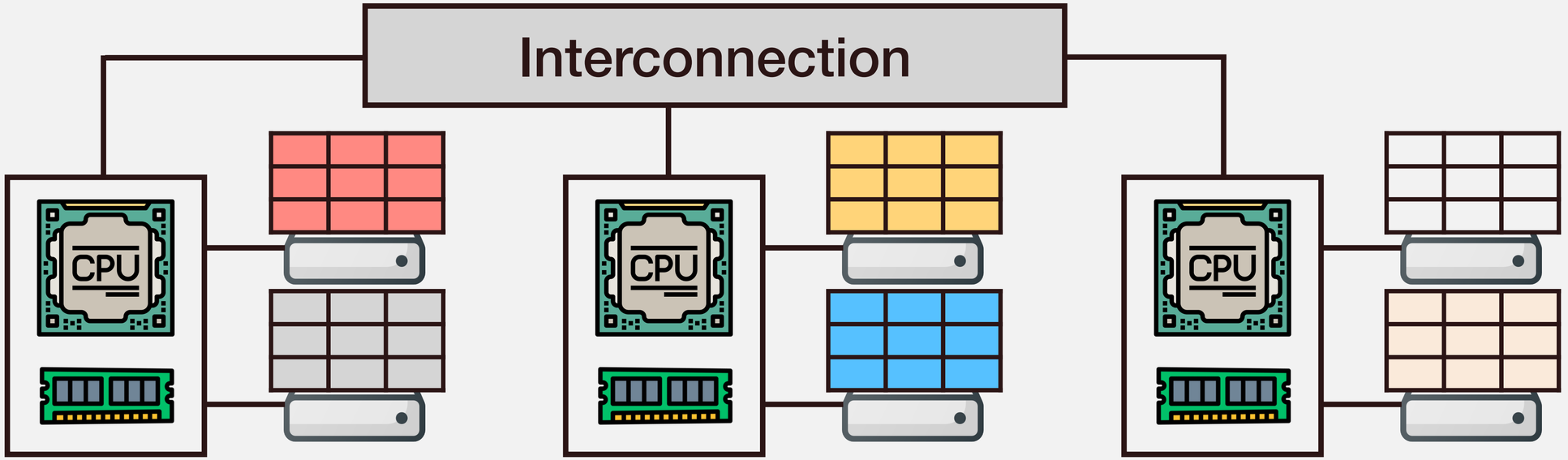
Shared-Nothing Architecture



Shared-Nothing Architecture



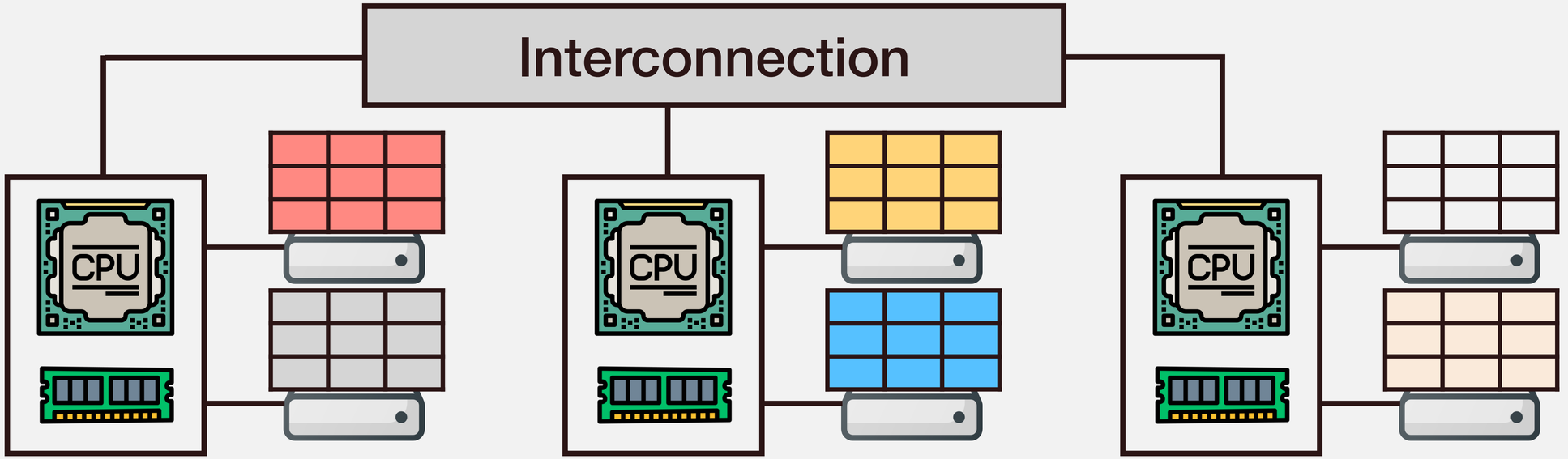
Shared-Nothing Architecture



PROS

- Horizontal Scalability
- Simple and elegant design

Shared-Nothing Architecture



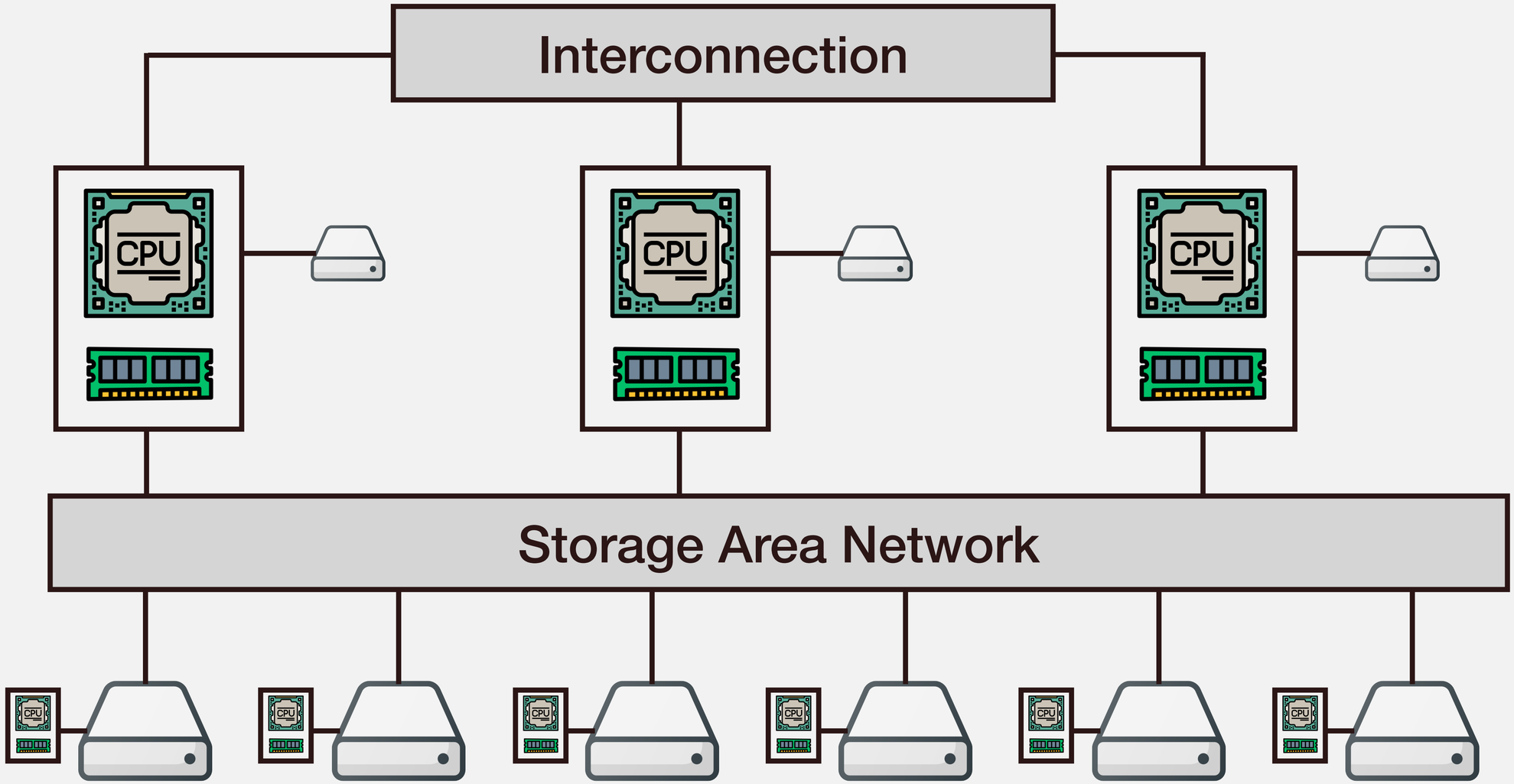
PROS

- Horizontal Scalability
- Simple and elegant design

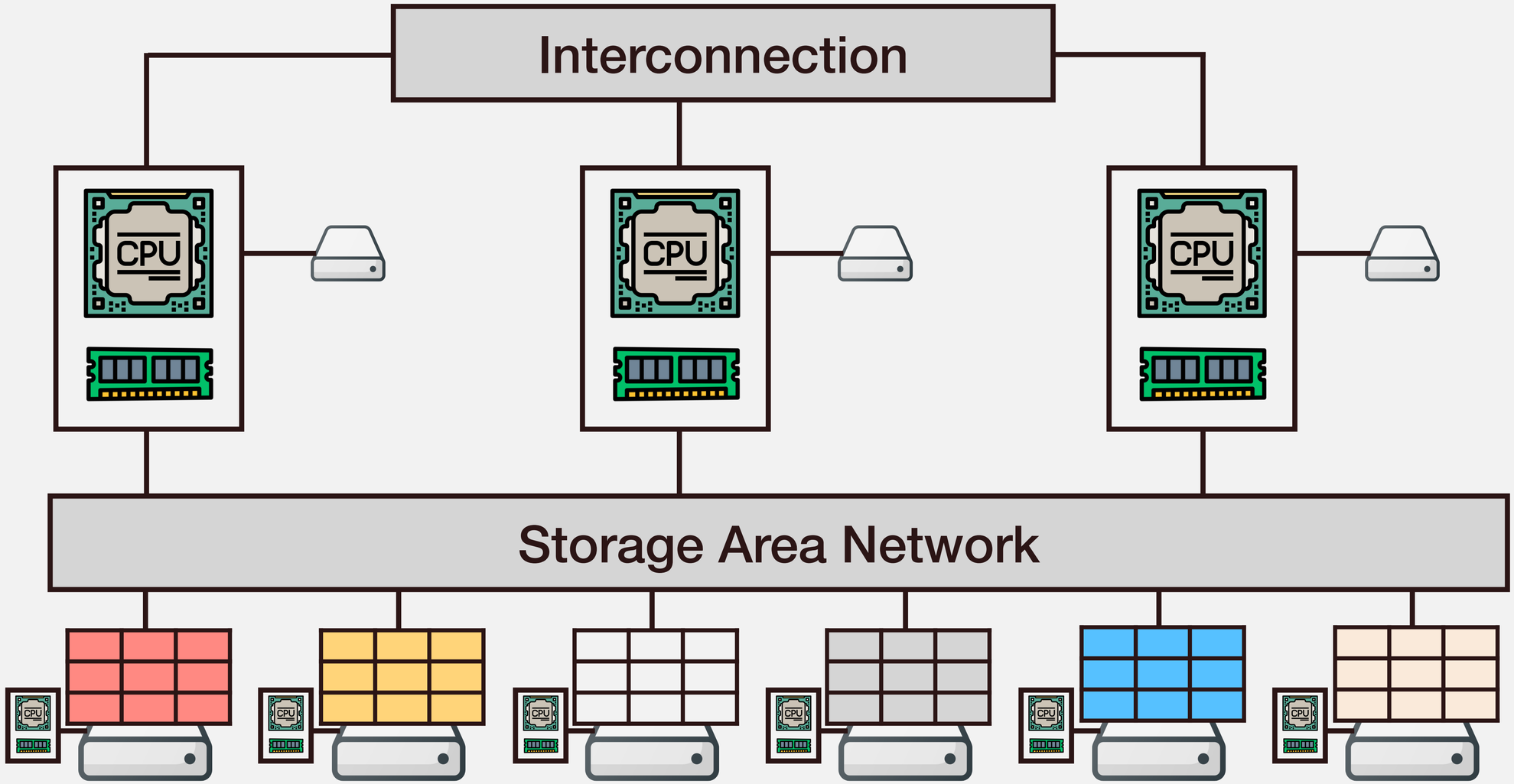
CONS

- Cross-machine operations
- Data redistribution at cluster resizing

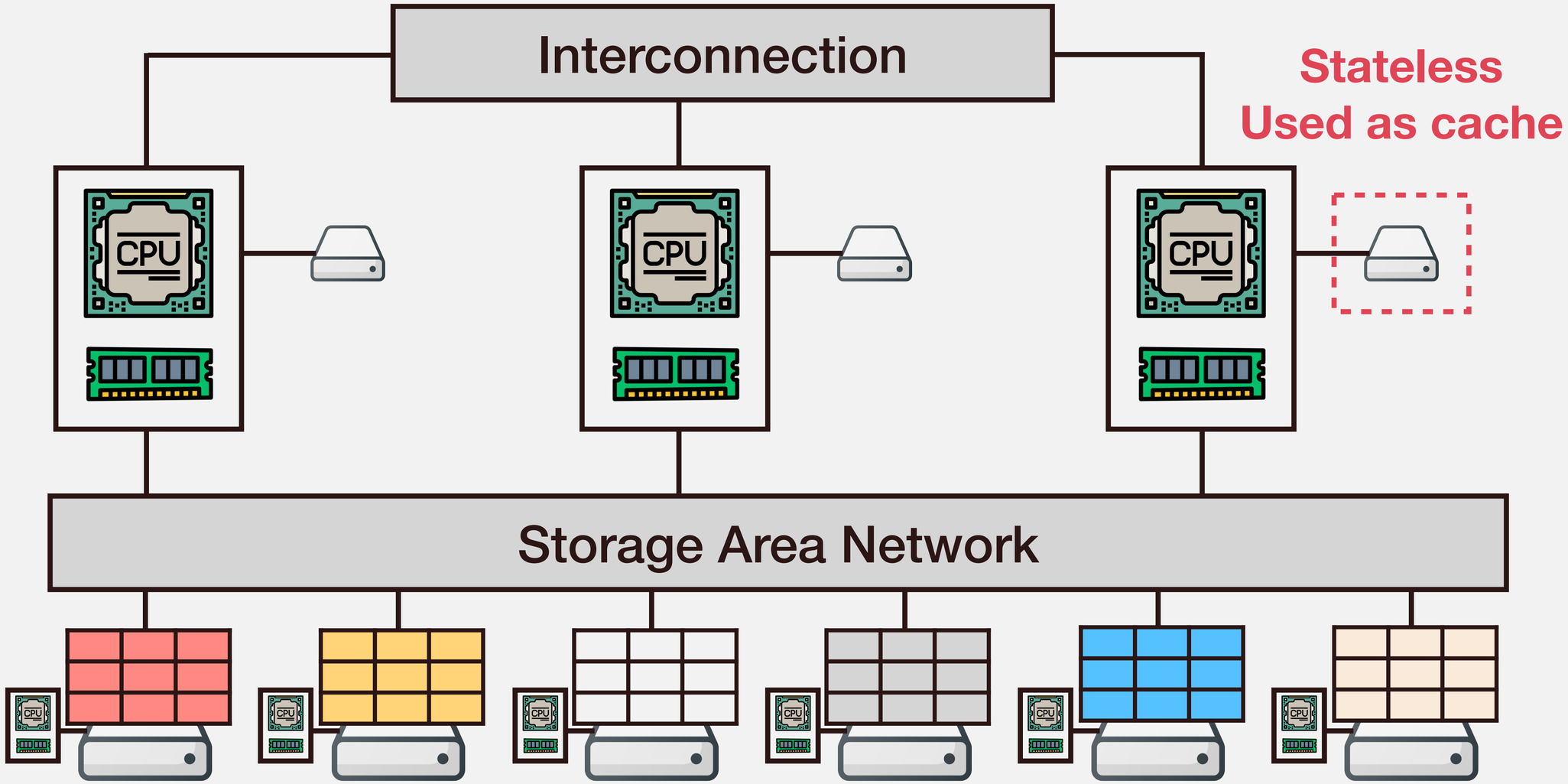
Separating Compute and Storage Nodes



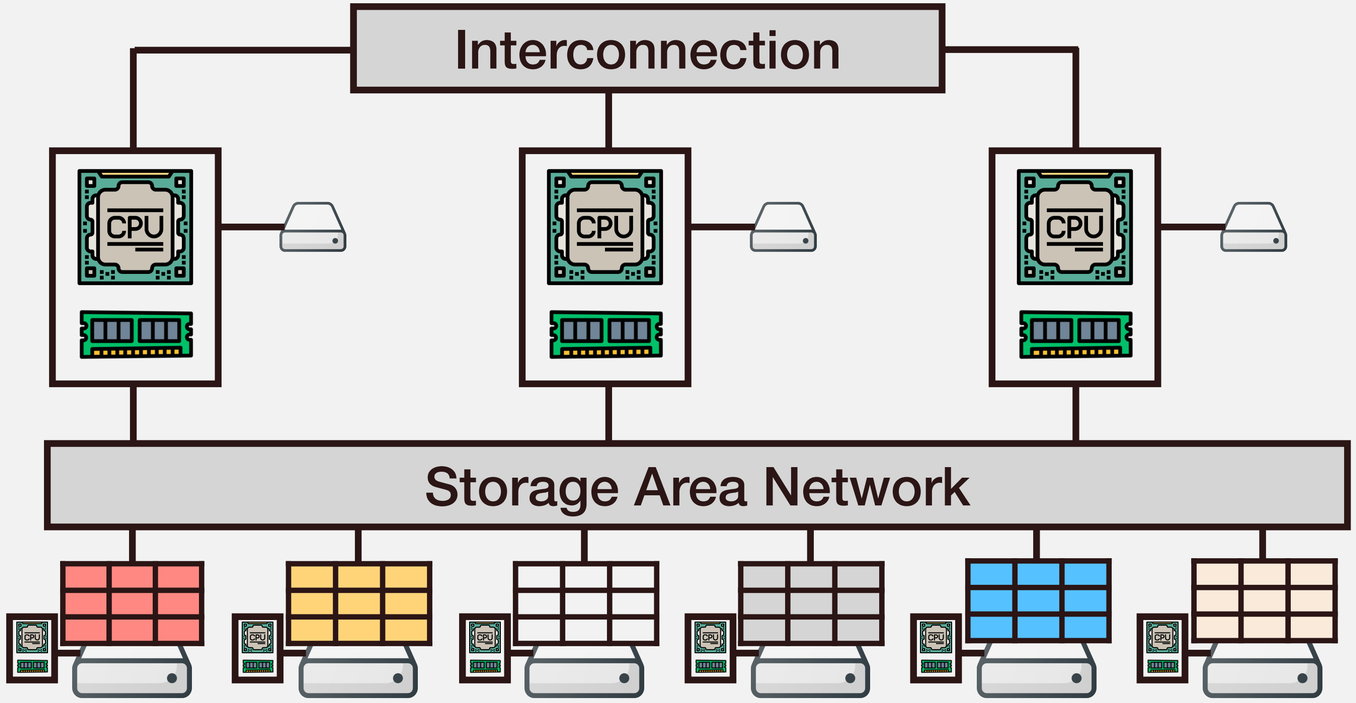
Separating Compute and Storage Nodes



Separating Compute and Storage Nodes



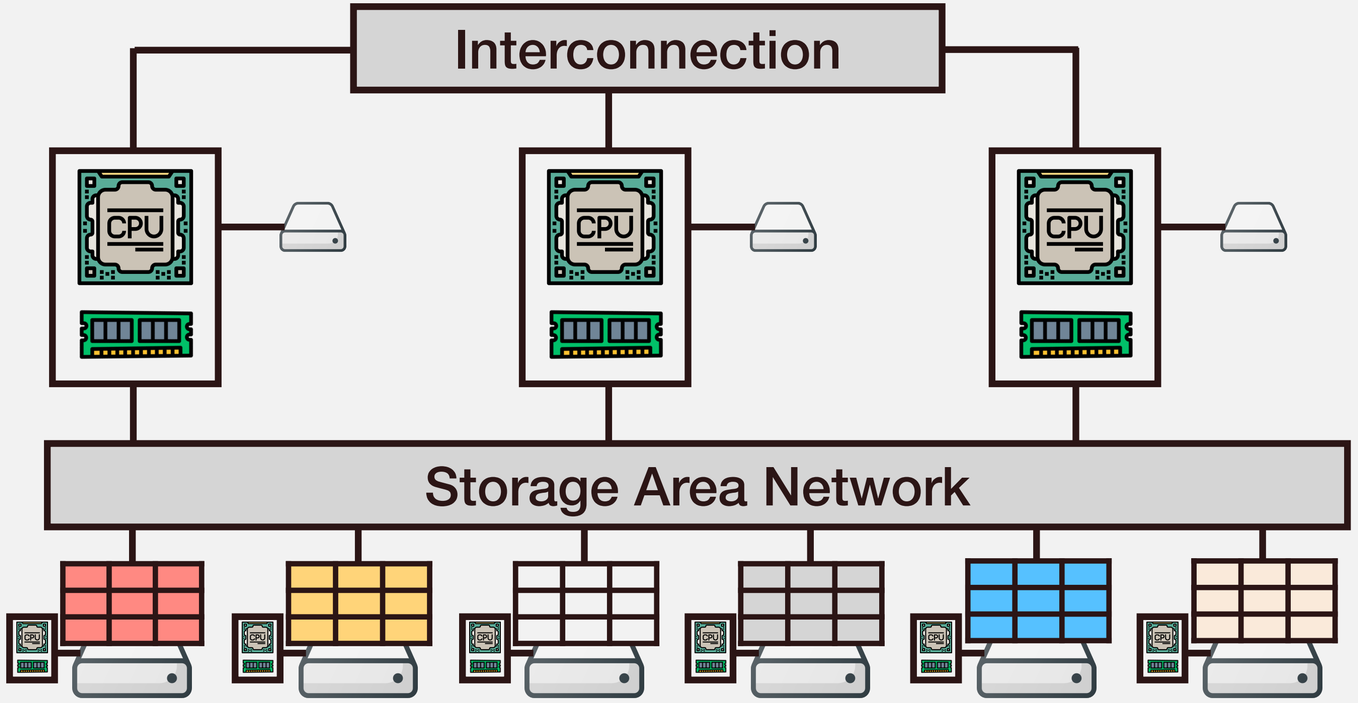
Separating Compute and Storage Nodes



Key Benefit:

Compute and storage can **scale independently**

Separating Compute and Storage Nodes

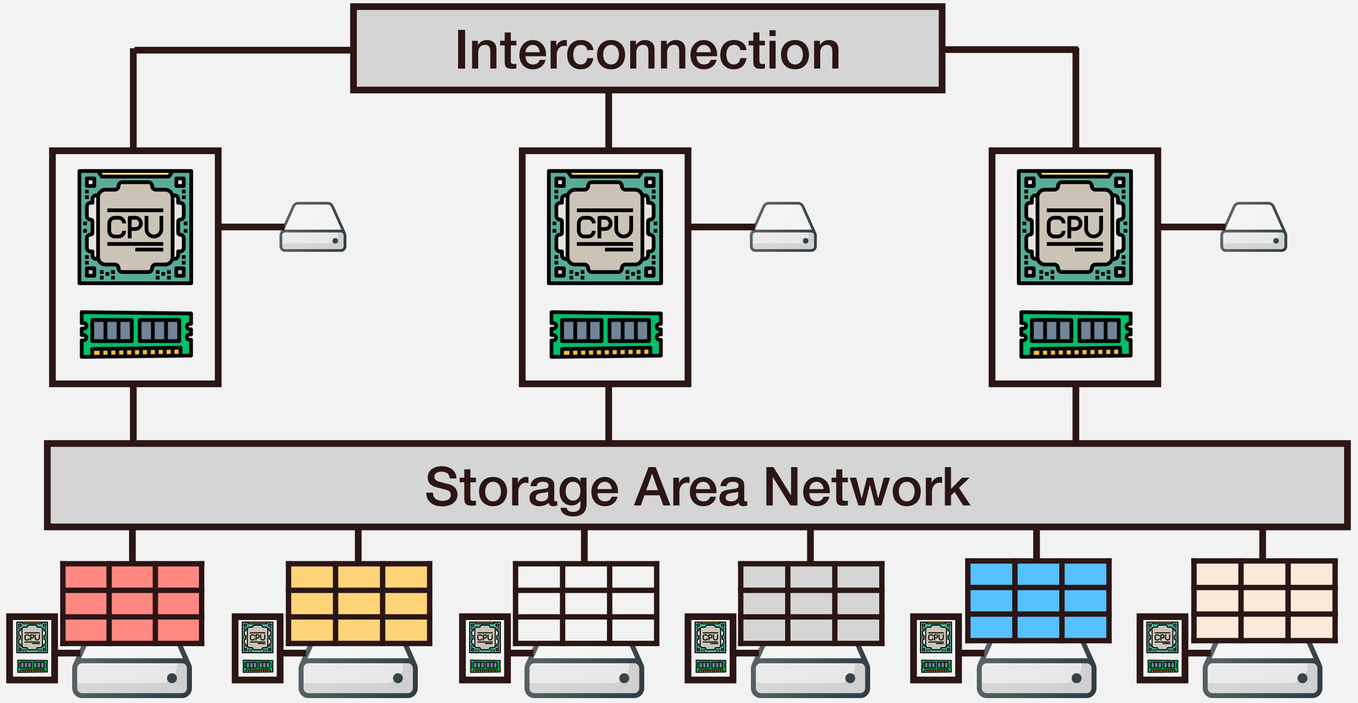


Key Benefit:

Compute and storage can **scale independently**

Performance depends on network latency and bandwidth

Separating Compute and Storage Nodes



Key Benefit:

Compute and storage can **scale independently**

Performance depends on network latency and bandwidth

Memory Disaggregation ?

Separating Compute and Storage Nodes

“未来的数据库可能就像拼乐高积木一样，每一个组件，存储、计算都可以从架构角度进行解耦，从而像拼乐高一样根据客户的应用需求快速组合起来”

Memory Disaggregation ?

Today

- ➜ Cloud Computing & Cloud Services
- ➜ Compute and Storage Disaggregation
- ➜ **Cloud Data Warehouses**

Online Transaction Processing (OLTP)

Example: online shopping, stock market transactions, …

```
SELECT *
FROM ShoppingCart
WHERE customerID = …
```

```
INSERT INTO Orders
VALUES (…)
```

```
UPDATE Accounts
SET balance = …
WHERE customerID = …
```

Online Transaction Processing (OLTP)

Example: online shopping, stock market transactions, …

- Simple, short-lived transactions (ms)
- Only touch a small amount of data
- Insert- and update-heavy
- Few table joins
- Skewed access towards recent data
- Queries often predefined

```
SELECT *
FROM ShoppingCart
WHERE customerID = …
```

```
INSERT INTO Orders
VALUES (…)
```

```
UPDATE Accounts
SET balance = …
WHERE customerID = …
```

Online Transaction Processing (OLTP)

Example: online shopping, stock market transactions, …

- Simple, short-lived transactions (ms)
- Only touch a small amount of data
- Insert- and update-heavy
- Few table joins
- Skewed access towards recent data
- Queries often predefined

Large number of concurrent operations

```
SELECT *
FROM ShoppingCart
WHERE customerID = …
```

```
INSERT INTO Orders
VALUES (…)
```

```
UPDATE Accounts
SET balance = …
WHERE customerID = …
```

Online Analytical Processing (OLAP)

Example: data analytics, business report, …

```
SELECT
    l_orderkey,
    sum(l_extendedprice*(1-l_discount)) as revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = '[SEGMENT]'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < date '[DATE]'
    and l_shipdate > date '[DATE]'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, o_orderdate
LIMIT 10
```

Online Analytical Processing (OLAP)

Example: data analytics, business report, …

- Complex, long-running aggregations
- Large table scans
- Mostly reads with periodic batch inserts
- Often joins multiple tables
- Historical data
- Queries often ad hoc

```
SELECT
    l_orderkey,
    sum(l_extendedprice*(1-l_discount)) as revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = '[SEGMENT]'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < date '[DATE]'
    and l_shipdate > date '[DATE]'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, o_orderdate
LIMIT 10
```

Online Analytical Processing (OLAP)

Example: data analytics, business report, …

- Complex, long-running aggregations
- Large table scans
- Mostly reads with periodic batch inserts
- Often joins multiple tables
- Historical data
- Queries often ad hoc

Heavy compute on large volume of data

```
SELECT
    l_orderkey,
    sum(l_extendedprice*(1-l_discount)) as revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = '[SEGMENT]'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < date '[DATE]'
    and l_shipdate > date '[DATE]'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, o_orderdate
LIMIT 10
```

Row-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Row-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Tuple-by-tuple Storage

105 Alice 18 1000	102 Bob 25 2000	104 Charlie 18 3000	104 David 18 1500	103 Emily 20 2500

Row-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros

- Fast tuple insertion/deletion
- Fast SELECT *

Tuple-by-tuple Storage

105 Alice 18 1000	102 Bob 25 2000	104 Charlie 18 3000	104 David 18 1500	103 Emily 20 2500

Row-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros

- Fast tuple insertion/deletion
- Fast SELECT *

Cons

- **SELECT avg**(balance) **FROM** T **GROUP BY** age

Tuple-by-tuple Storage

105 Alice 18 1000	102 Bob 25 2000	104 Charlie 18 3000	104 David 18 1500	103 Emily 20 2500

Row-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros

- Fast tuple insertion/deletion
- Fast SELECT *

Cons

- **SELECT avg**(balance) **FROM** T **GROUP BY** age
- Reading useless data: wasting I/O

Tuple-by-tuple Storage

105 Alice 18 1000	102 Bob 25 2000	104 Charlie 18 3000	104 David 18 1500	103 Emily 20 2500

Row-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros **Ideal for OLTP**

- Fast tuple insertion/deletion
- Fast SELECT *

Cons

- **SELECT avg**(balance) **FROM** T **GROUP BY** age
- Reading useless data: wasting I/O

Tuple-by-tuple Storage

105 Alice 18 1000	102 Bob 25 2000	104 Charlie 18 3000	104 David 18 1500	103 Emily 20 2500

Column-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Column-by-Column Storage

105 102 104 101 103	Alice Bob Charlie David Emily	18 25 18 18 20	1000 2000 3000 1500 2500
id	**name**	**age**	**balance**

Column-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros

- Only scan relevant attributes
- Fast and efficient query processing

Column-by-Column Storage

105 102 104 101 103	Alice Bob Charlie David Emily	18 25 18 18 20	1000 2000 3000 1500 2500
id	**name**	**age**	**balance**

Column-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros

- Only scan relevant attributes
- Fast and efficient query processing

Cons

- **INSERT INTO** T **VALUES** (a, b, c, d, …)

Column-by-Column Storage

105 102 104 101 103	Alice Bob Charlie David Emily	18 25 18 18 20	1000 2000 3000 1500 2500
id	**name**	**age**	**balance**

Column-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros

- Only scan relevant attributes
- Fast and efficient query processing

Cons

- **INSERT INTO** T **VALUES** (a, b, c, d, …)
- **SELECT** * **…**

Column-by-Column Storage

105 102 104 101 103	Alice Bob Charlie David Emily	18 25 18 18 20	1000 2000 3000 1500 2500

id **name** **age** **balance**

Column-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros

- Only scan relevant attributes
- Fast and efficient query processing

Cons

- **INSERT INTO** T **VALUES** (a, b, c, d, …)
- **SELECT * …**
- Extra work in tuple splitting and stitching

Column-by-Column Storage

105 102 104 101 103	Alice Bob Charlie David Emily	18 25 18 18 20	1000 2000 3000 1500 2500

id **name** **age** **balance**

Column-Store

Table is 2D, but storage is 1D array

id	name	age	balance
105	Alice	18	1000
102	Bob	25	2000
104	Charlie	18	3000
101	David	18	1500
103	Emily	20	2500

Pros **Ideal for OLAP**

- Only scan relevant attributes
- Fast and efficient query processing

Cons

- **INSERT INTO** T **VALUES** (a, b, c, d, …)
- **SELECT * …**
- Extra work in tuple splitting and stitching

Column-by-Column Storage

105 102 104 101 103	Alice Bob Charlie David Emily	18 25 18 18 20	1000 2000 3000 1500 2500
id	**name**	**age**	**balance**

Column-Store is Everywhere

➜ Becomes popular in the late 2000s

- Vertica (C-Store), MonetDB, VectorWise

➜ Every major data warehouse today

- Teradata, Amazon Redshift, Snowflake, Google BigQuery, ClickHouse, Greenplum …

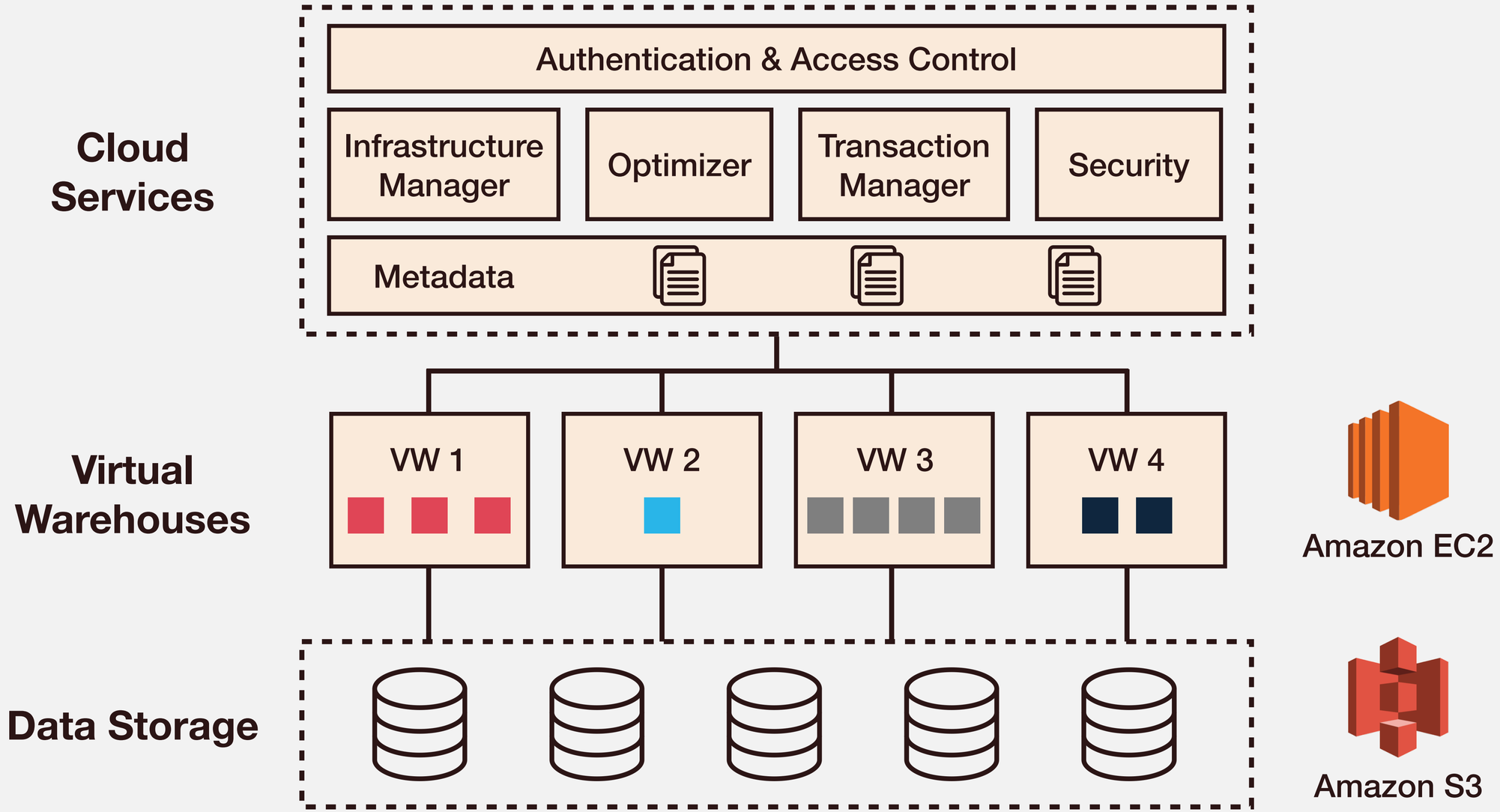
➜ Traditional row-stores intending to support OLAP-type queries

- Oracle 12c, SQL Server, IBM DB2 BLU

Snowflake

- ➜ The Snowflake Elastic Data Warehouse
  - Target analytical queries to support business intelligence (BI)
  - Founded at 2012, growing fast, largest software IPO (2020) ever
- ➜ Pure SaaS
  - Nothing to install, always on, always up-to-date
  - Ease of use, only pay for what you use
- ➜ Multi-Cloud Support

Snowflake Architecture



Snowflake Architecture

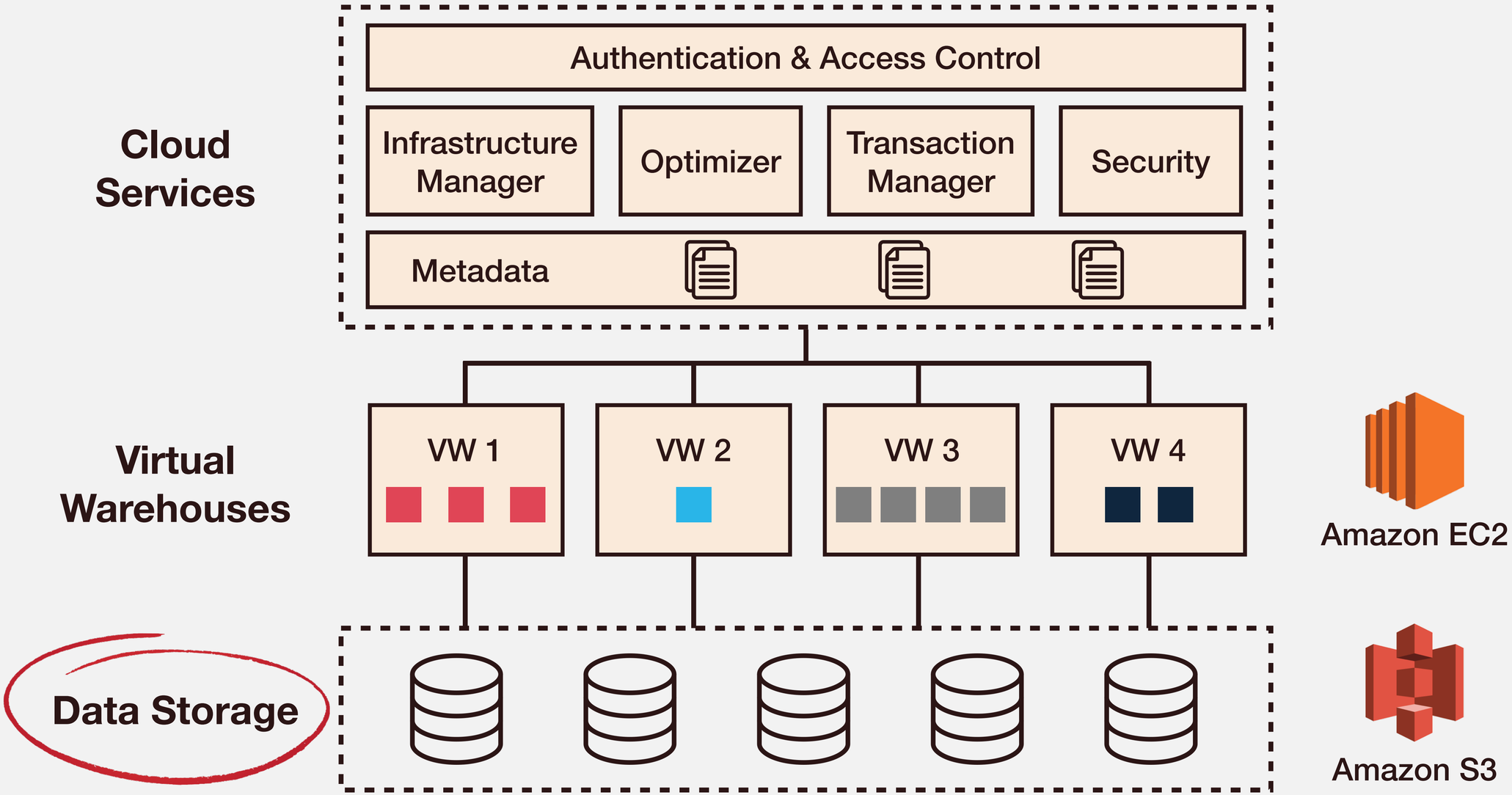
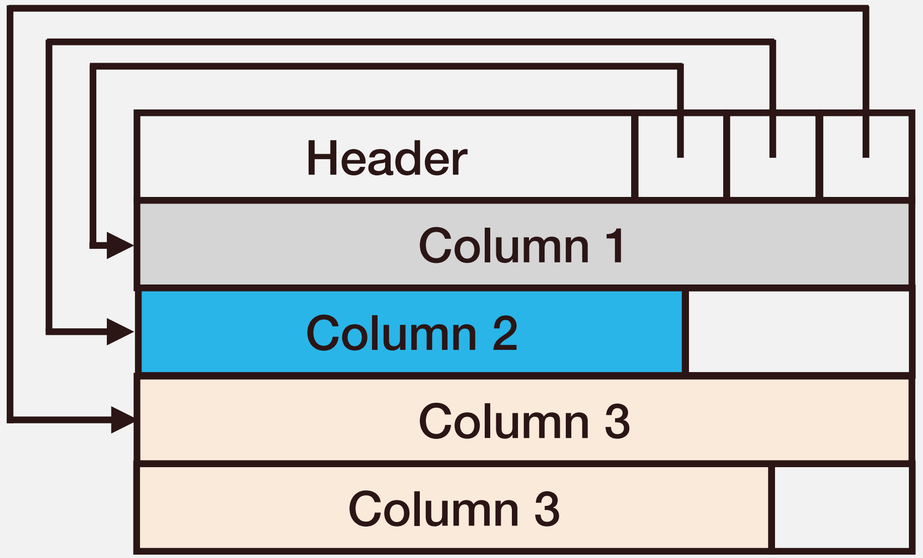


Table File Format

snowflake

- Tables are horizontally partitioned
  - Micro-partition: size = 10s MB, natural ingestion order
- Hybrid columnar (PAX) format



Queries first read header

Heavily Compressed

Hybrid Columnar Format

➜ **Tables are horizontally portioned into files, where each file is stored in columnar format**

Pure Columnar

File 1

name
Alice
David
Bob
Emily
Bob
Emily
Bob
Charlie
Emily

File 2

age
18
18
23
20
18
22
19
20
19

File 3

balance
1000
1500
2000
2500
3000
3500
4000
4500
5000

Hybrid Columnar Format

➜ **Tables are horizontally portioned into files, where each file is stored in columnar format**

Pure Columnar

File 1

name
Alice
David
Bob
Emily
Bob
Emily
Bob
Charlie
Emily

File 2

age
18
18
23
20
18
22
19
20
19

File 3

balance
1000
1500
2000
2500
3000
3500
4000
4500
5000

Hybrid Columnar (a.k.a PAX)

File 1

name	age	balance
Alice	18	1000
David	18	1500
Bob	23	2000

File 2

name	age	balance
Emily	20	2500
Bob	18	3000
Emily	22	3500

File 3

name	age	balance
Bob	19	4000
Charlie	20	4500
Emily	19	5000

Hybrid Columnar Format

➜ **Tables are horizontally portioned into files, where each file is stored in columnar format**

Pure Columnar

File 1

name
Alice
David
Bob
Emily
Bob
Emily
Bob
Charlie
Emily

File 2

age
18
18
23
20
18
22
19
20
19

File 3

balance
1000
1500
2000
2500
3000
3500
4000
4500
5000

Hybrid Columnar (a.k.a PAX)

File 1

name	age	balance
Alice	18	1000
David	18	1500
Bob	23	2000

A disk page in the original proposal

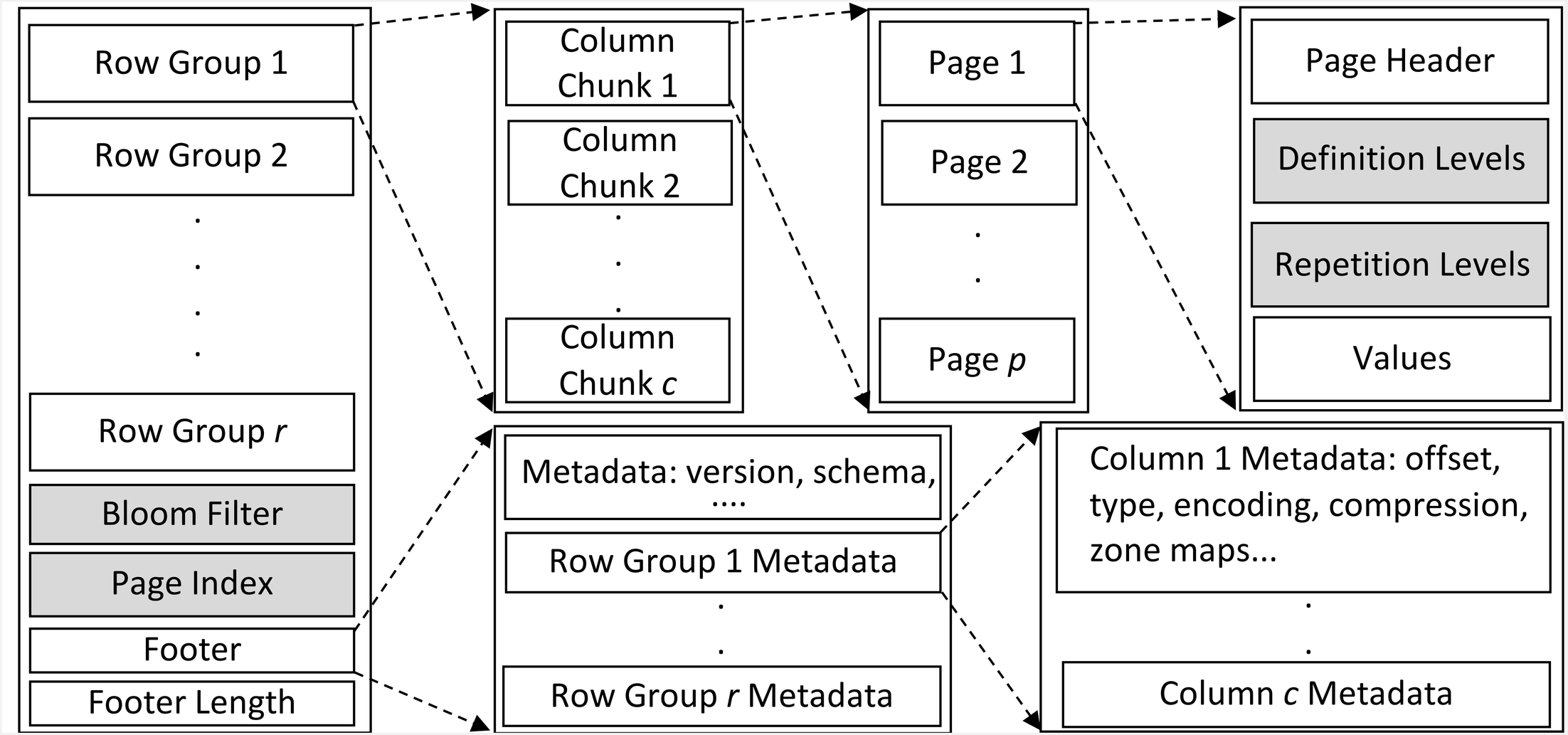
File 2

name	age	balance
Emily	20	2500
Bob	18	3000
Emily	22	3500

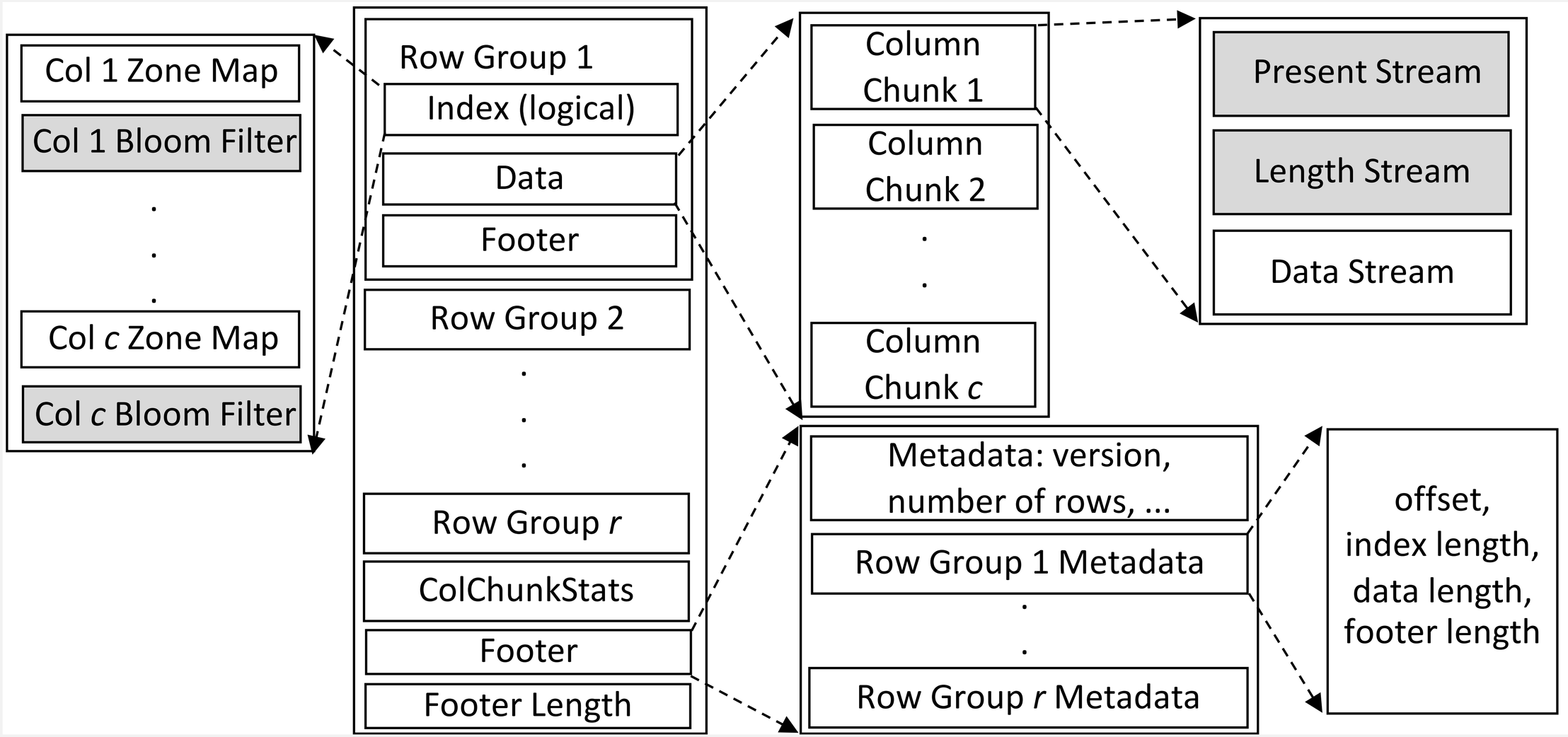
File 3

name	age	balance
Bob	19	4000
Charlie	20	4500
Emily	19	5000

Apache Parquet



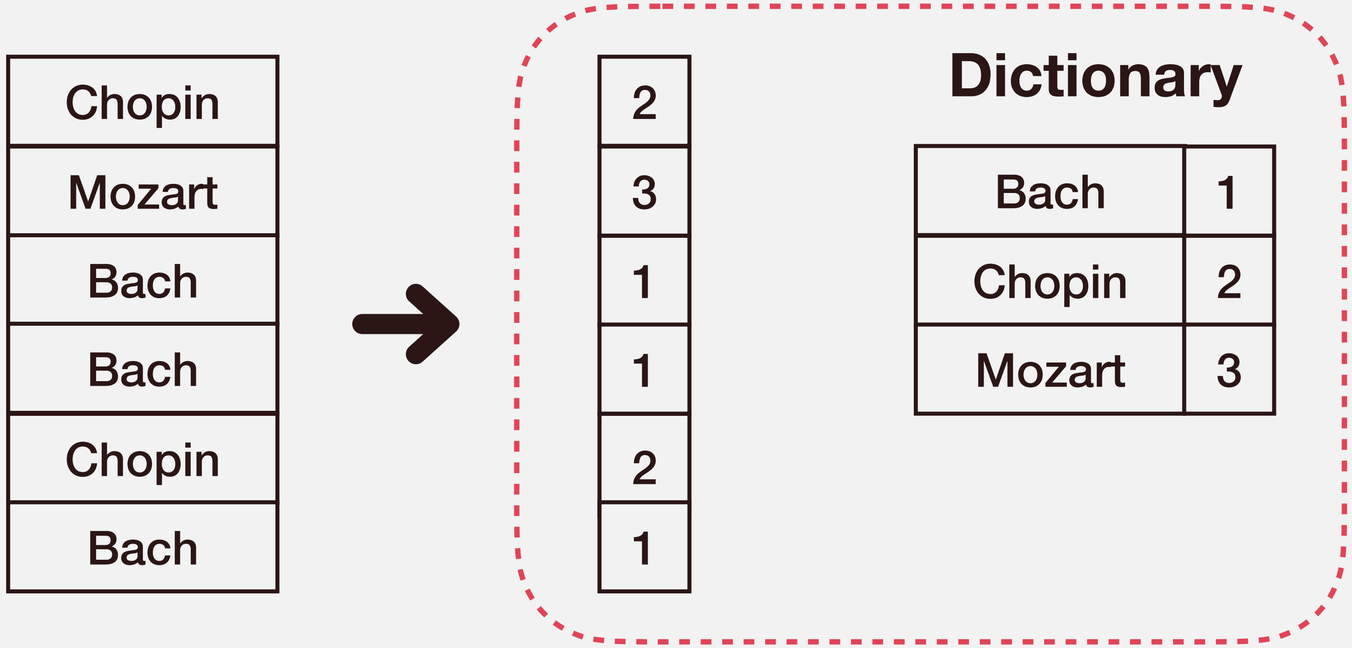
Apache ORC


Col 1 Zone Map
Col 1 Bloom Filter
Col c Zone Map
Col c Bloom Filter
Row Group 1
Index (logical)
Data
Footer
Row Group 2
Row Group r
ColChunkStats
Footer
Footer Length
Column Chunk 1
Column Chunk 2
Column Chunk c
Present Stream
Length Stream
Data Stream
Metadata: version, number of rows, ...
Row Group 1 Metadata
Row Group r Metadata
offset, index length, data length, footer length

Dictionary Encoding

➜ **Replace frequent patterns with smaller codes**

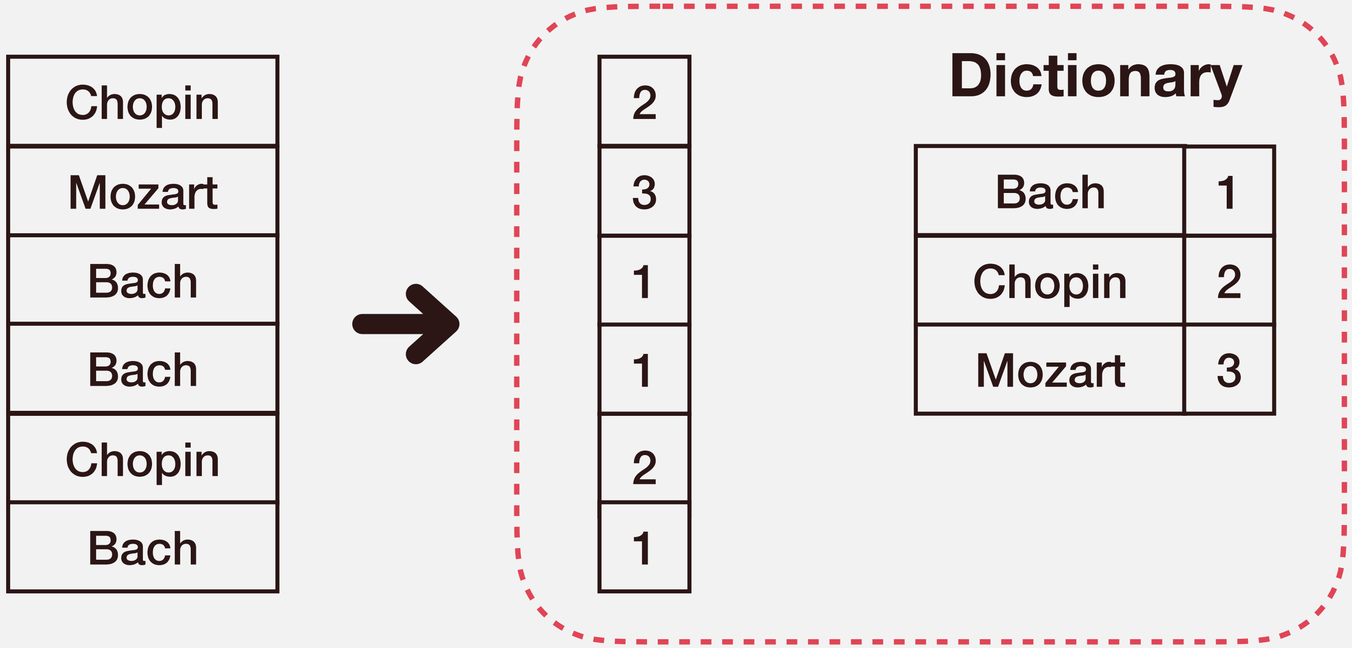
- One of the most prevalent compression schemes in DBMSs (both row-store and column-store)
- Usually applied on String or Categorical columns



Dictionary Encoding

➜ **Replace frequent patterns with smaller codes**

- One of the most prevalent compression schemes in DBMSs (both row-store and column-store)
- Usually applied on String or Categorical columns



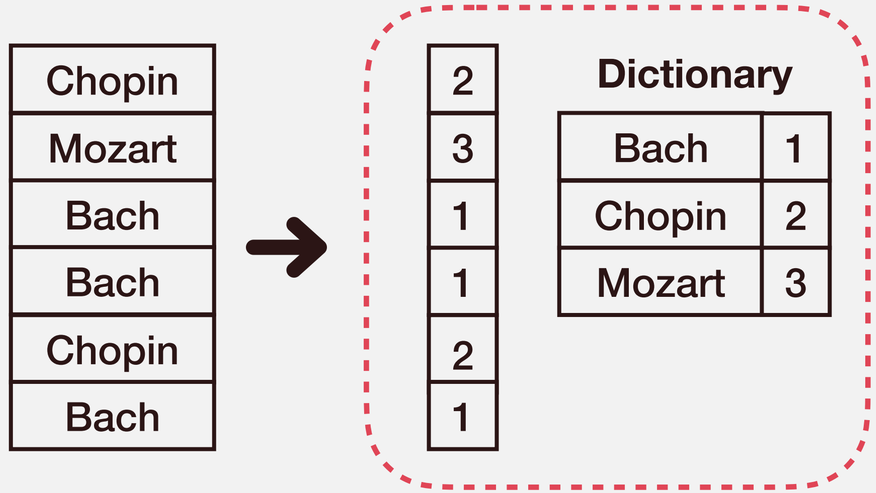
Parquet — Limits dictionary by **physical size**

Apache orc™ — Limits dictionary by **NDV ratio**

Dictionary Encoding

➔ **Preserve value ordering?**

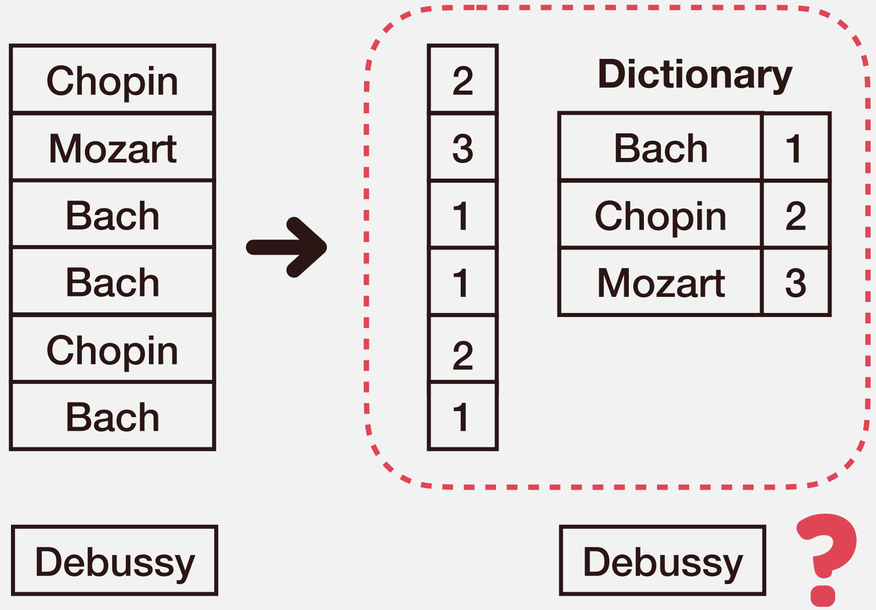
- Benefit: applying range predicate without decompression



Dictionary Encoding

➔ **Preserve value ordering?**

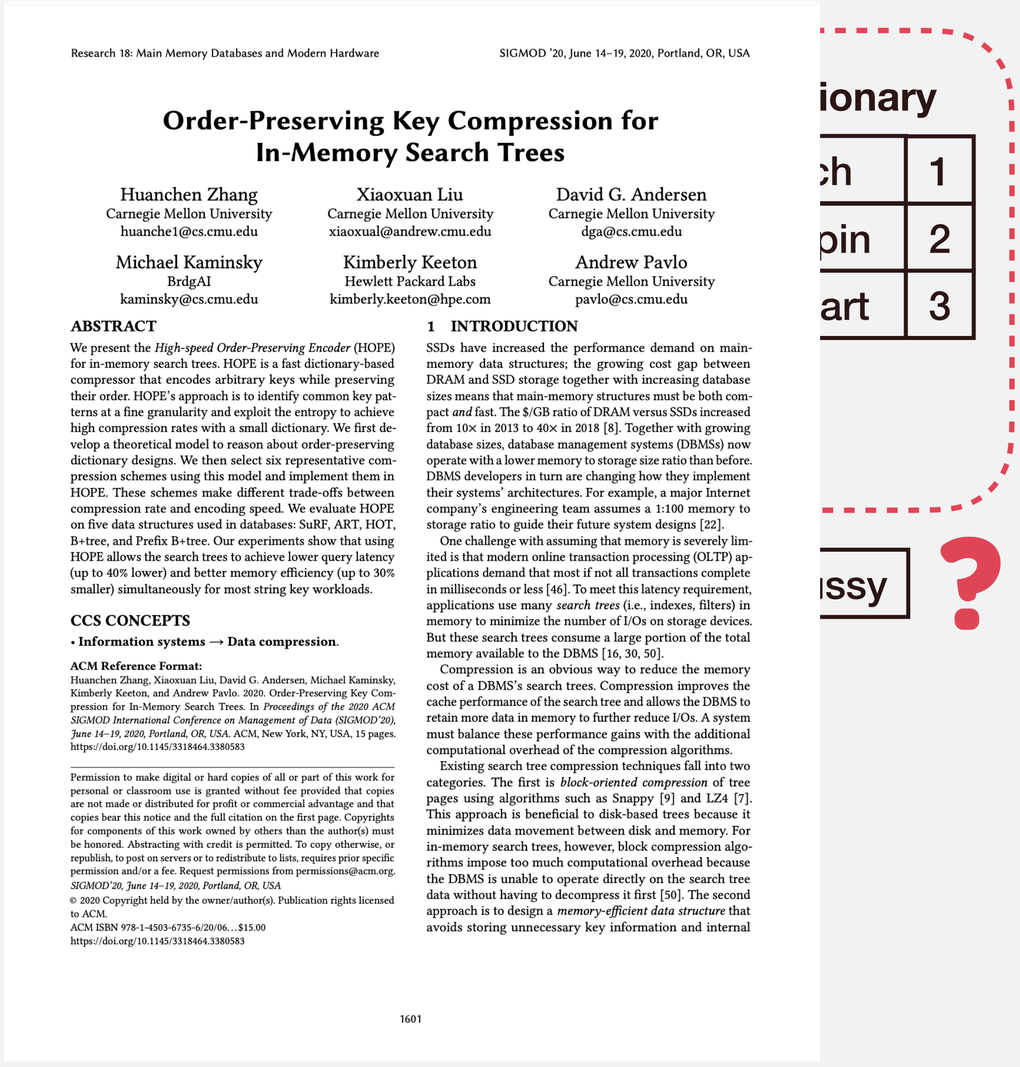
- Benefit: applying range predicate without decompression



Dictionary Encoding

➔ **Preserve value ordering?**

- Benefit: applying range predicate without decompression
- How to minimize/avoid dictionary rebuild?
  - Research! e.g., HOPE [SIGMOD'20]



Research 18: Main Memory Databases and Modern Hardware SIGMOD '20, June 14–19, 2020, Portland, OR, USA

Order-Preserving Key Compression for In-Memory Search Trees

Huanchen Zhang
Carnegie Mellon University
huanche1@cs.cmu.edu

Xiaoxuan Liu
Carnegie Mellon University
xiaoxual@andrew.cmu.edu

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

Michael Kaminsky
BrdgAI
kaminsky@cs.cmu.edu

Kimberly Keeton
Hewlett Packard Labs
kimberly.keeton@hpe.com

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

We present the *High-speed Order-Preserving Encoder* (HOPE) for in-memory search trees. HOPE is a fast dictionary-based compressor that encodes arbitrary keys while preserving their order. HOPE's approach is to identify common key patterns at a fine granularity and exploit the entropy to achieve high compression rates with a small dictionary. We first develop a theoretical model to reason about order-preserving dictionary designs. We then select six representative compression schemes using this model and implement them in HOPE. These schemes make different trade-offs between compression rate and encoding speed. We evaluate HOPE on five data structures used in databases: SuRF, ART, HOT, B+tree, and Prefix B+tree. Our experiments show that using HOPE allows the search trees to achieve lower query latency (up to 40% lower) and better memory efficiency (up to 30% smaller) simultaneously for most string key workloads.

CCS CONCEPTS

• **Information systems** → **Data compression**.

ACM Reference Format:
Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380583

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380583

1 INTRODUCTION

SSDs have increased the performance demand on main-memory data structures; the growing cost gap between DRAM and SSD storage together with increasing database sizes means that main-memory structures must be both compact *and* fast. The $/GB ratio of DRAM versus SSDs increased from 10× in 2013 to 40× in 2018 [8]. Together with growing database sizes, database management systems (DBMSs) now operate with a lower memory to storage size ratio than before. DBMS developers in turn are changing how they implement their systems' architectures. For example, a major Internet company's engineering team assumes a 1:100 memory to storage ratio to guide their future system designs [22].

One challenge with assuming that memory is severely limited is that modern online transaction processing (OLTP) applications demand that most if not all transactions complete in milliseconds or less [46]. To meet this latency requirement, applications use many *search trees* (i.e., indexes, filters) in memory to minimize the number of I/Os on storage devices. But these search trees consume a large portion of the total memory available to the DBMS [16, 30, 50].

Compression is an obvious way to reduce the memory cost of a DBMS's search trees. Compression improves the cache performance of the search tree and allows the DBMS to retain more data in memory to further reduce I/Os. A system must balance these performance gains with the additional computational overhead of the compression algorithms.

Existing search tree compression techniques fall into two categories. The first is *block-oriented compression* of tree pages using algorithms such as Snappy [9] and LZ4 [7]. This approach is beneficial to disk-based trees because it minimizes data movement between disk and memory. For in-memory search trees, however, block compression algorithms impose too much computational overhead because the DBMS is unable to operate directly on the search tree data without having to decompress it first [50]. The second approach is to design a *memory-efficient data structure* that avoids storing unnecessary key information and internal

1601

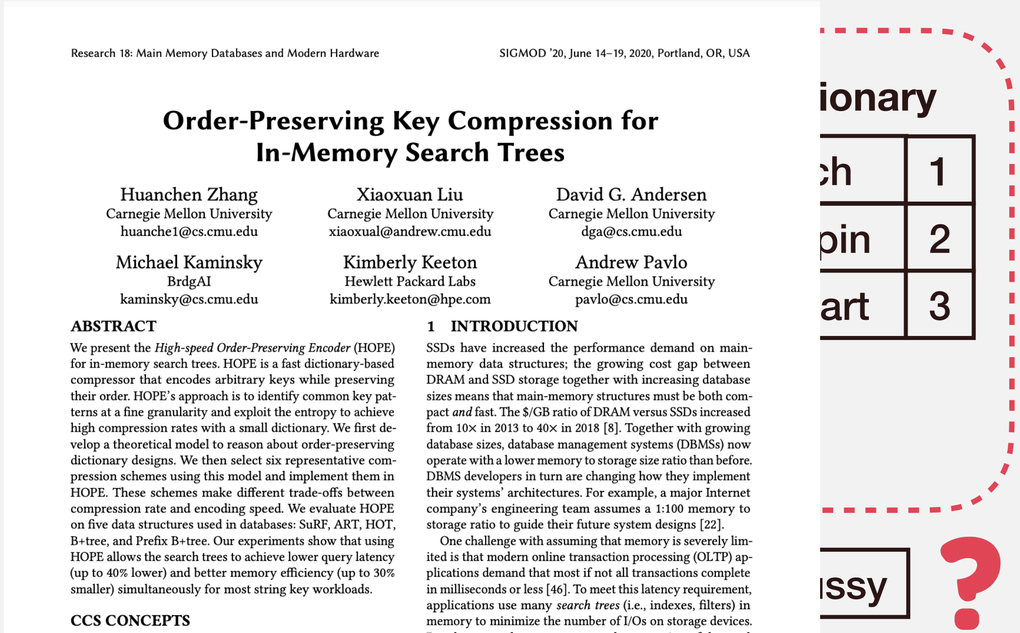
Dictionary Encoding

➜ **Preserve value ordering?**

- Benefit: applying range predicate without decompression
- How to minimize/avoid dictionary rebuild?
  - Research! e.g., HOPE [SIGMOD'20]

➜ **Dictionary scope?**

- Block-level? Table-level? Across tables?



Order-Preserving Key Compression for In-Memory Search Trees

Huanchen Zhang
Carnegie Mellon University
huanche1@cs.cmu.edu

Xiaoxuan Liu
Carnegie Mellon University
xiaoxual@andrew.cmu.edu

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

Michael Kaminsky
BrdgAI
kaminsky@cs.cmu.edu

Kimberly Keeton
Hewlett Packard Labs
kimberly.keeton@hpe.com

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

We present the *High-speed Order-Preserving Encoder* (HOPE) for in-memory search trees. HOPE is a fast dictionary-based compressor that encodes arbitrary keys while preserving their order. HOPE's approach is to identify common key patterns at a fine granularity and exploit the entropy to achieve high compression rates with a small dictionary. We first develop a theoretical model to reason about order-preserving dictionary designs. We then select six representative compression schemes using this model and implement them in HOPE. These schemes make different trade-offs between compression rate and encoding speed. We evaluate HOPE on five data structures used in databases: SuRF, ART, HOT, B+tree, and Prefix B+tree. Our experiments show that using HOPE allows the search trees to achieve lower query latency (up to 40% lower) and better memory efficiency (up to 30% smaller) simultaneously for most string key workloads.

CCS CONCEPTS

• **Information systems** → **Data compression**.

ACM Reference Format:
Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380583

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380583

1 INTRODUCTION

SSDs have increased the performance demand on main-memory data structures; the growing cost gap between DRAM and SSD storage together with increasing database sizes means that main-memory structures must be both compact *and* fast. The $/GB ratio of DRAM versus SSDs increased from 10× in 2013 to 40× in 2018 [8]. Together with growing database sizes, database management systems (DBMSs) now operate with a lower memory to storage size ratio than before. DBMS developers in turn are changing how they implement their systems' architectures. For example, a major Internet company's engineering team assumes a 1:100 memory to storage ratio to guide their future system designs [22].

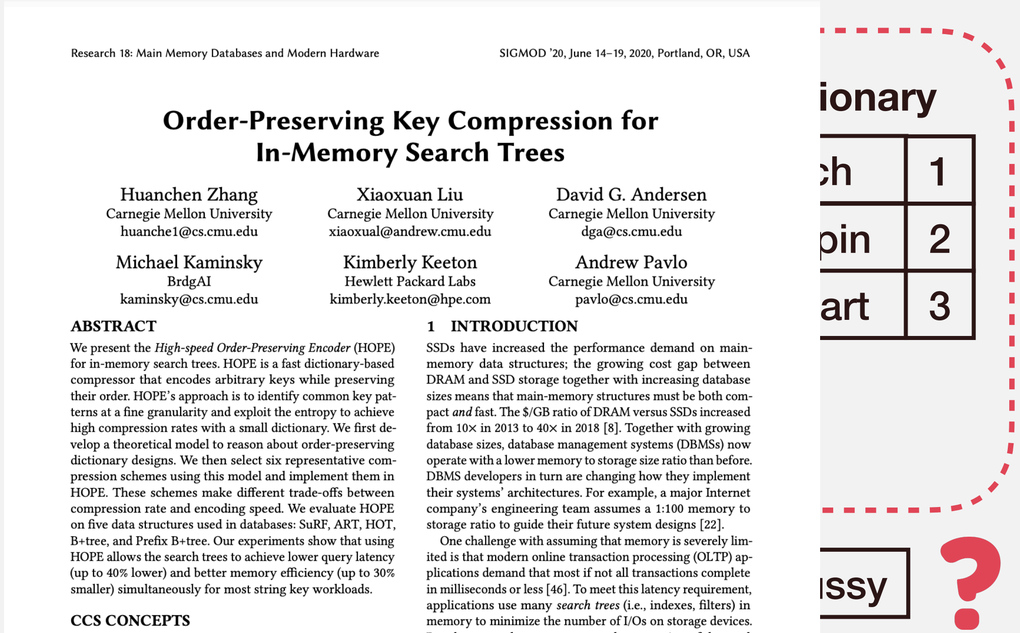
One challenge with assuming that memory is severely limited is that modern online transaction processing (OLTP) applications demand that most if not all transactions complete in milliseconds or less [46]. To meet this latency requirement, applications use many *search trees* (i.e., indexes, filters) in memory to minimize the number of I/Os on storage devices. But these search trees consume a large portion of the total memory available to the DBMS [16, 30, 50].

Compression is an obvious way to reduce the memory cost of a DBMS's search trees. Compression improves the cache performance of the search tree and allows the DBMS to retain more data in memory to further reduce I/Os. A system must balance these performance gains with the additional computational overhead of the compression algorithms.

Existing search tree compression techniques fall into two categories. The first is *block-oriented compression* of tree pages using algorithms such as Snappy [9] and LZ4 [7]. This approach is beneficial to disk-based trees because it minimizes data movement between disk and memory. For in-memory search trees, however, block compression algorithms impose too much computational overhead because the DBMS is unable to operate directly on the search tree data without having to decompress it first [50]. The second approach is to design a *memory-efficient data structure* that avoids storing unnecessary key information and internal

Dictionary Encoding

- ➔ **Preserve value ordering?**
  - Benefit: applying range predicate without decompression
  - How to minimize/avoid dictionary rebuild?
    - Research! e.g., HOPE [SIGMOD'20]
- ➔ **Dictionary scope?**
  - Block-level? Table-level? Across tables?
- ➔ **Data Structure for the dictionary?**
  - Array? Hash table? B+tree?



Order-Preserving Key Compression for In-Memory Search Trees

Huanchen Zhang
Carnegie Mellon University
huanche1@cs.cmu.edu

Xiaoxuan Liu
Carnegie Mellon University
xiaoxual@andrew.cmu.edu

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

Michael Kaminsky
BrdgAI
kaminsky@cs.cmu.edu

Kimberly Keeton
Hewlett Packard Labs
kimberly.keeton@hpe.com

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

We present the *High-speed Order-Preserving Encoder* (HOPE) for in-memory search trees. HOPE is a fast dictionary-based compressor that encodes arbitrary keys while preserving their order. HOPE's approach is to identify common key patterns at a fine granularity and exploit the entropy to achieve high compression rates with a small dictionary. We first develop a theoretical model to reason about order-preserving dictionary designs. We then select six representative compression schemes using this model and implement them in HOPE. These schemes make different trade-offs between compression rate and encoding speed. We evaluate HOPE on five data structures used in databases: SuRF, ART, HOT, B+tree, and Prefix B+tree. Our experiments show that using HOPE allows the search trees to achieve lower query latency (up to 40% lower) and better memory efficiency (up to 30% smaller) simultaneously for most string key workloads.

CCS CONCEPTS

• **Information systems** → **Data compression**.

ACM Reference Format:
Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380583

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380583

1 INTRODUCTION

SSDs have increased the performance demand on main-memory data structures; the growing cost gap between DRAM and SSD storage together with increasing database sizes means that main-memory structures must be both compact *and* fast. The $/GB ratio of DRAM versus SSDs increased from 10× in 2013 to 40× in 2018 [8]. Together with growing database sizes, database management systems (DBMSs) now operate with a lower memory to storage size ratio than before. DBMS developers in turn are changing how they implement their systems' architectures. For example, a major Internet company's engineering team assumes a 1:100 memory to storage ratio to guide their future system designs [22].

One challenge with assuming that memory is severely limited is that modern online transaction processing (OLTP) applications demand that most if not all transactions complete in milliseconds or less [46]. To meet this latency requirement, applications use many *search trees* (i.e., indexes, filters) in memory to minimize the number of I/Os on storage devices. But these search trees consume a large portion of the total memory available to the DBMS [16, 30, 50].

Compression is an obvious way to reduce the memory cost of a DBMS's search trees. Compression improves the cache performance of the search tree and allows the DBMS to retain more data in memory to further reduce I/Os. A system must balance these performance gains with the additional computational overhead of the compression algorithms.

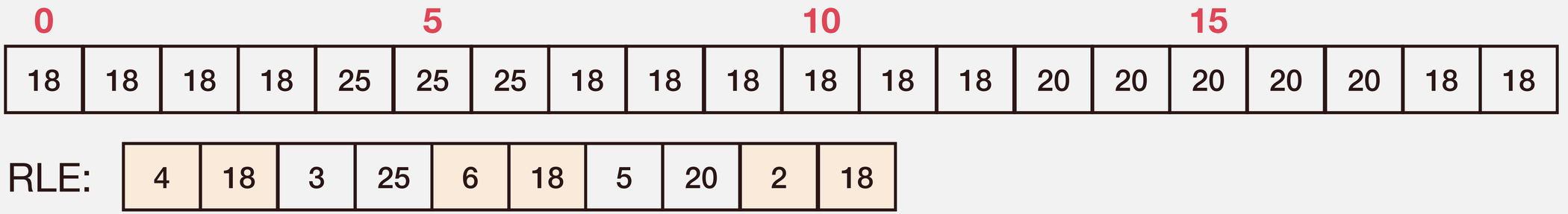
Existing search tree compression techniques fall into two categories. The first is *block-oriented compression* of tree pages using algorithms such as Snappy [9] and LZ4 [7]. This approach is beneficial to disk-based trees because it minimizes data movement between disk and memory. For in-memory search trees, however, block compression algorithms impose too much computational overhead because the DBMS is unable to operate directly on the search tree data without having to decompress it first [50]. The second approach is to design a *memory-efficient data structure* that avoids storing unnecessary key information and internal

Run-Length Encoding (RLE)

➜ **Represent consecutive repeated values using (length, value)**

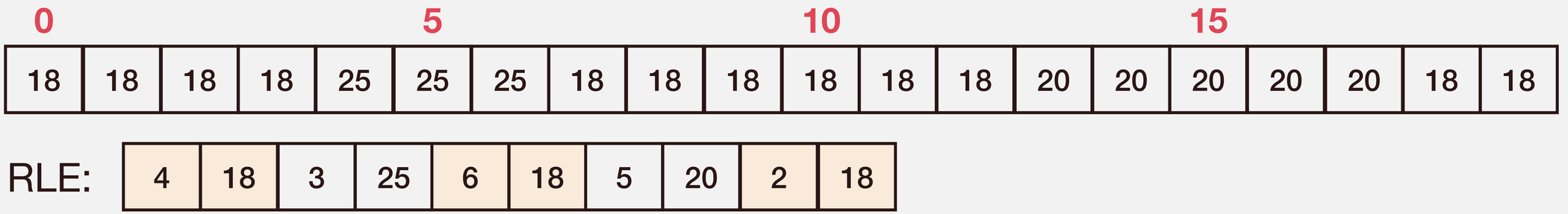
Run-Length Encoding (RLE)

➜ **Represent consecutive repeated values using (length, value)**



Run-Length Encoding (RLE)

➜ **Represent consecutive repeated values using (length, value)**

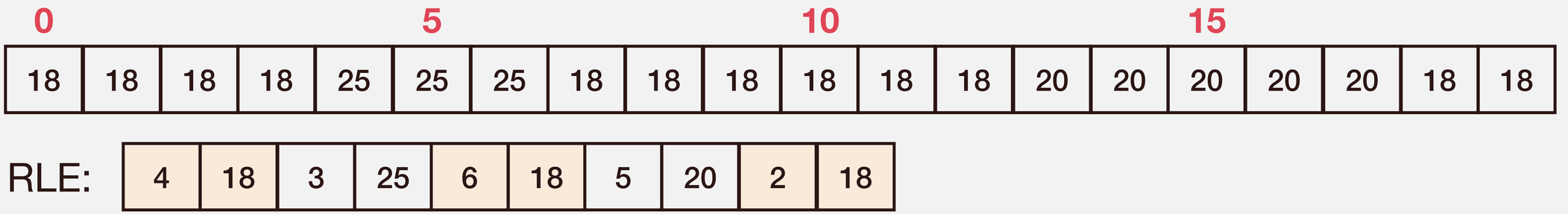


Works best when

- The column is pre-sorted
- The column contains a lot of repetition

Run-Length Encoding (RLE)

➜ **Represent consecutive repeated values using (length, value)**



Works best when

- The column is pre-sorted
- The column contains a lot of repetition

Parquet run-length ≥ 8

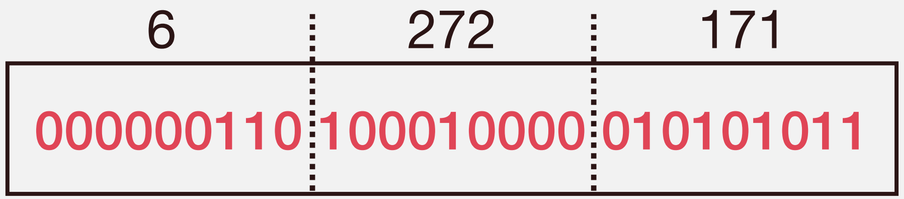
Apache ORC $3 \leq$ run-length ≤ 10

Bit-Packing

➜ **Store numbers using as few bits as possible**

Example: 32-bit integers

With Null Suppression



Delta Encoding

➔ **Record the difference between values**

1000001	1000002	1000011	1000005	1000008	1000009

Delta Encoding

➔ **Record the difference between values**

1000001	1000002	1000011	1000005	1000008	1000009

Delta-encoded

1000001	+1	+9	-6	+3	+1

fewer bits

Delta Encoding

➔ **Record the difference between values**

1000001	1000002	1000011	1000005	1000008	1000009

Delta-encoded

1000001	+1	+9	-6	+3	+1

fewer bits

Works best when

- The column is (almost) sorted
- The values are dense

} e.g., timestamps, temperature reading

Delta Encoding

➔ **Record the difference between values**

1000001	1000002	1000011	1000005	1000008	1000009

Delta-encoded

1000001	+1	+9	-6	+3	+1

fewer bits

Works best when

- The column is (almost) sorted
- The values are dense

} e.g., timestamps, temperature reading

Downside

- Random access becomes slow

Frame of Reference (FOR)

➔ **Record the deltas to the first (or min) value in the block**

1000001	1000002	1000011	1000005	1000008	1000009

FOR-encoded

1000001	+1	+10	+4	+7	+8

Frame of Reference (FOR)

➔ **Record the deltas to the first (or min) value in the block**

1000001	1000002	1000011	1000005	1000008	1000009

FOR-encoded

1000001	+1	+10	+4	+7	+8

Compared to Delta Encoding

Frame of Reference (FOR)

➔ **Record the deltas to the first (or min) value in the block**

1000001	1000002	1000011	1000005	1000008	1000009

FOR-encoded

1000001	+1	+10	+4	+7	+8

Compared to Delta Encoding

+ Faster random access
+ Enables vectorized processing
- Potentially worse compression rate

Value Sequence Compression

Value Sequence: $v_1, v_2, v_3, \ldots, v_n$

➜ What's the theoretical bound?

Value Sequence Compression

Value Sequence: $v_1, v_2, v_3, \ldots, v_n$

➜ What's the theoretical bound?

- If i.i.d. random variables: **Shannon's Entropy**

Value Sequence Compression

Value Sequence: $v_1, v_2, v_3, \ldots, v_n$

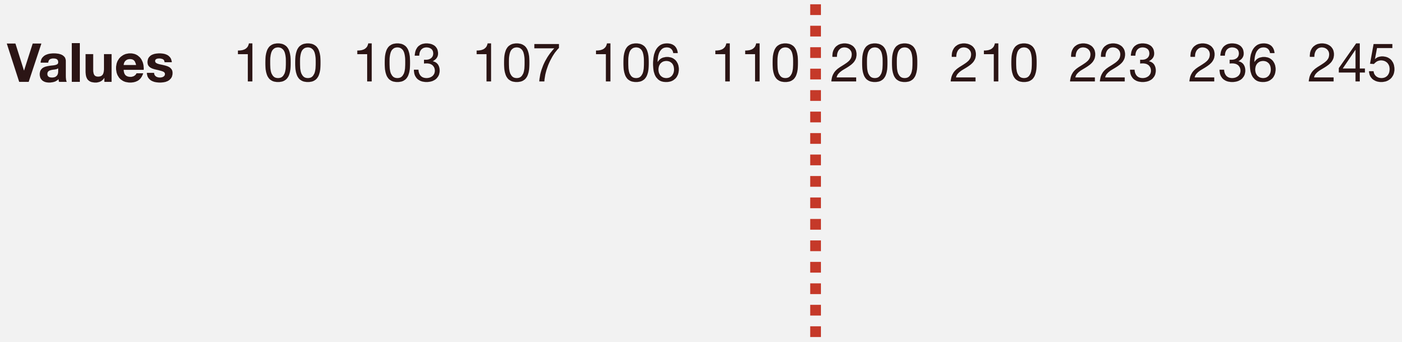
➜ What's the theoretical bound?

- If i.i.d. random variables: **Shannon's Entropy**
- If values are serially correlated: **Kolmogorov Complexity**

Learned Data Compression

Values 100 103 107 106 110 200 210 223 236 245

Learned Data Compression



Learned Data Compression

Values	100	103	107	106	110	200	210	223	236	245
Models			$2.3x + 98.1$					$11.6x + 188$		

Learned Data Compression

Values	100	103	107	106	110	200	210	223	236	245
Models	$2.3x + 98.1$					$11.6x + 188$				
Deltas	0	0	2	-1	0	0	-1	0	2	-1

Learned Data Compression

Values	100	103	107	106	110	200	210	223	236	245
Models	$2.3x + 98.1$					$11.6x + 188$				
Deltas	0	0	2	-1	0	0	-1	0	2	-1

Values $\overrightarrow{v}_{[0,n)} = (v_0, \ldots, v_{n-1})$

Partitions $\overrightarrow{v}_{[k_0=0,k_1)} \quad \overrightarrow{v}_{[k_1,k_2)} \quad \cdots \quad \overrightarrow{v}_{[k_{m-1},k_m=n)}$

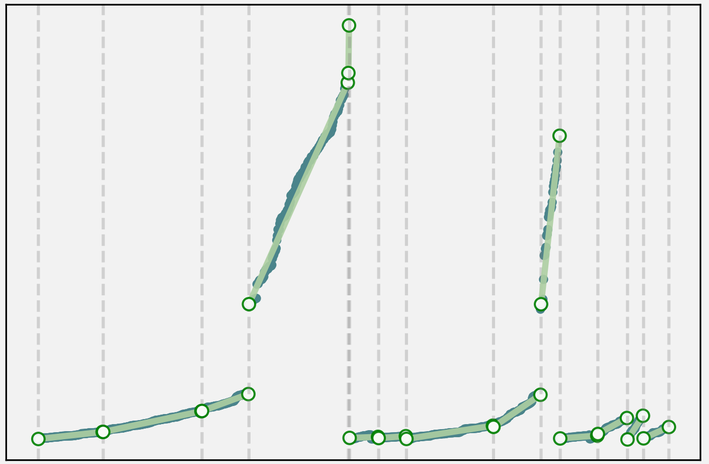
Models $\mathscr{F}_0 \quad \mathscr{F}_1 \quad \mathscr{F}_{m-1}$

Deltas $\Delta_i = v_i - \mathscr{F}_j(v_i), \text{ for } v_i \in \overrightarrow{v}_{[k_j,k_{j+1})}$

$$min \sum_{j=0}^{m-1} (\|\mathscr{F}_j\| + (k_{j+1} - k_j)(\max_{i=k_j}^{k_{j+1}-1} \lceil \log_2 \delta_i \rceil))$$

Learned Data Compression

Values	100	103	107	106	110	200	210	223	236	245
Models	$2.3x + 98.1$					$11.6x + 188$				
Deltas	0	0	2	-1	0	0	-1	0	2	-1



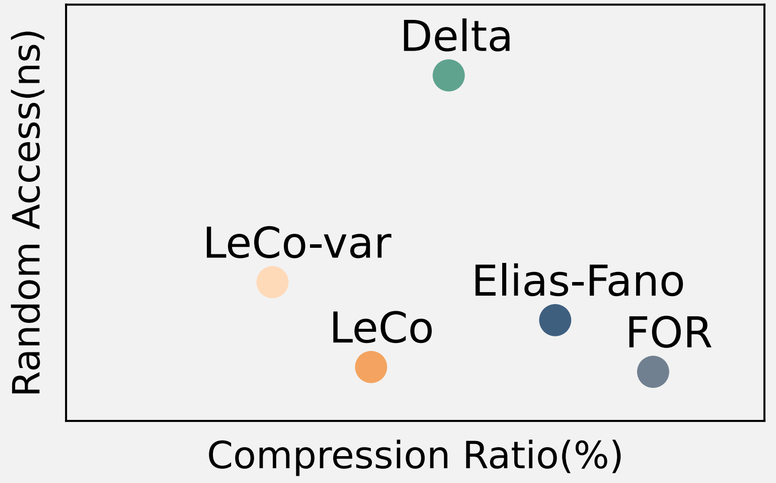
Values $\overrightarrow{v}_{[0,n)} = (v_0, \ldots, v_{n-1})$

Partitions $\overrightarrow{v}_{[k_0=0,k_1)} \quad \overrightarrow{v}_{[k_1,k_2)} \quad \cdots \quad \overrightarrow{v}_{[k_{m-1},k_m=n)}$

Models $\mathscr{F}_0 \quad \mathscr{F}_1 \quad \mathscr{F}_{m-1}$

Deltas $\Delta_i = v_i - \mathscr{F}_j(v_i), \text{ for } v_i \in \overrightarrow{v}_{[k_j,k_{j+1})}$

$$min \sum_{j=0}^{m-1} (\|\mathscr{F}_j\| + (k_{j+1} - k_j)(\max_{i=k_j}^{k_{j+1}-1} \lceil \log_2 \delta_i \rceil))$$



Toward Next-Gen Columnar Format

➔ **Lesson 1**: Dictionary Encoding is the most effective lightweight encoding algorithm for a columnar storage format because most real-world data have low NDV ratios.

Toward Next-Gen Columnar Format

- **Lesson 1**: Dictionary Encoding is the most effective lightweight encoding algorithm for a columnar storage format because most real-world data have low NDV ratios.

- **Lesson 2**: It is important to keep the encoding scheme simple in a columnar storage format to guarantee a competitive scan + decoding performance.

Toward Next-Gen Columnar Format

- **Lesson 1**: Dictionary Encoding is the most effective lightweight encoding algorithm for a columnar storage format because most real-world data have low NDV ratios.
- **Lesson 2**: It is important to keep the encoding scheme simple in a columnar storage format to guarantee a competitive scan + decoding performance.
- **Lesson 3**: A columnar storage format should enable block compression cautiously on modern hardware because the bottleneck of query processing is shifting from storage to computation.

Toward Next-Gen Columnar Format

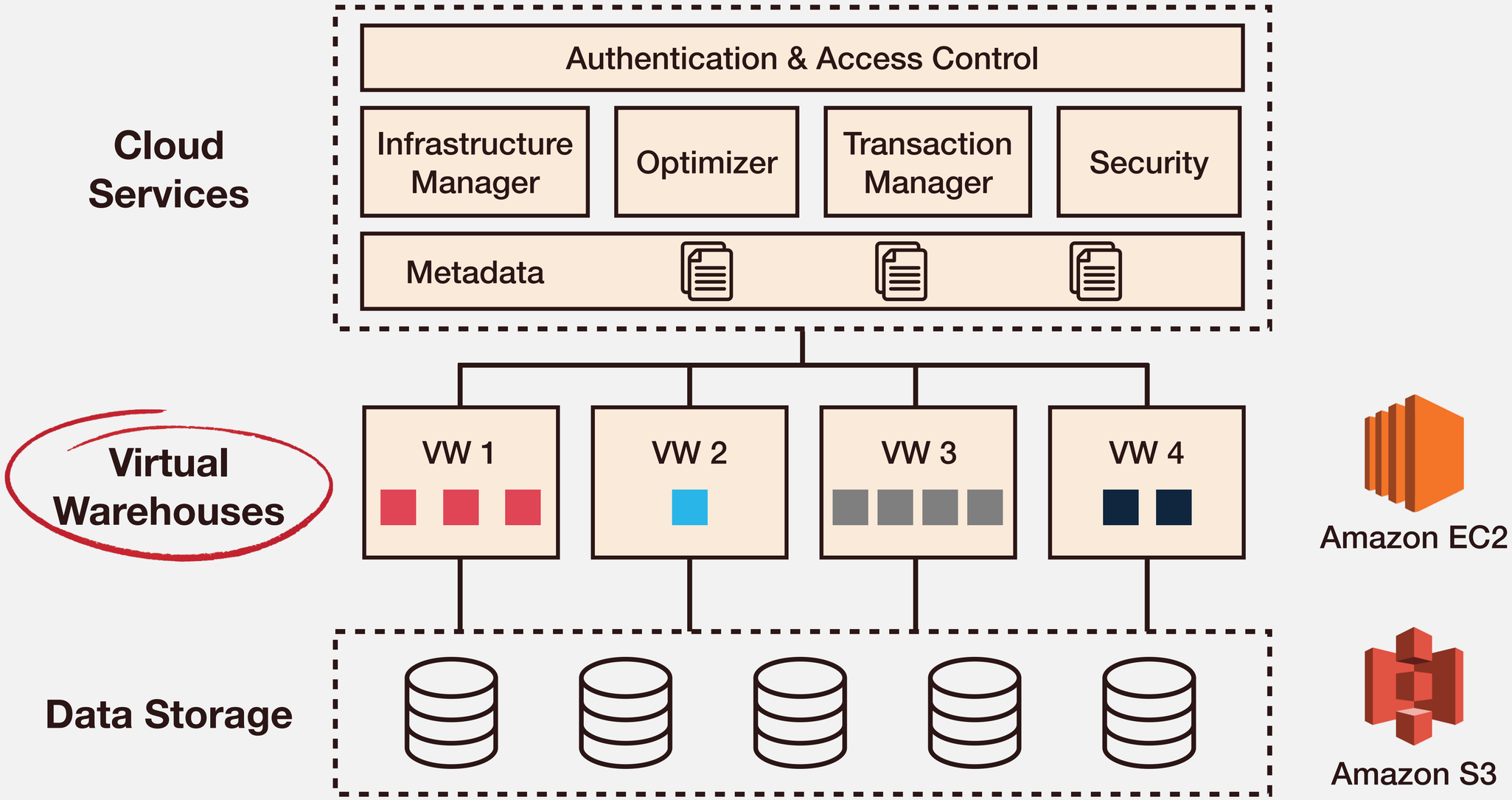
- **Lesson 1**: Dictionary Encoding is the most effective lightweight encoding algorithm for a columnar storage format because most real-world data have low NDV ratios.
- **Lesson 2**: It is important to keep the encoding scheme simple in a columnar storage format to guarantee a competitive scan + decoding performance.
- **Lesson 3**: A columnar storage format should enable block compression cautiously on modern hardware because the bottleneck of query processing is shifting from storage to computation.
- **Lesson 4**: The metadata layout in a columnar storage format should optimize for fewer random probes, especially with cloud storage.

Toward Next-Gen Columnar Format

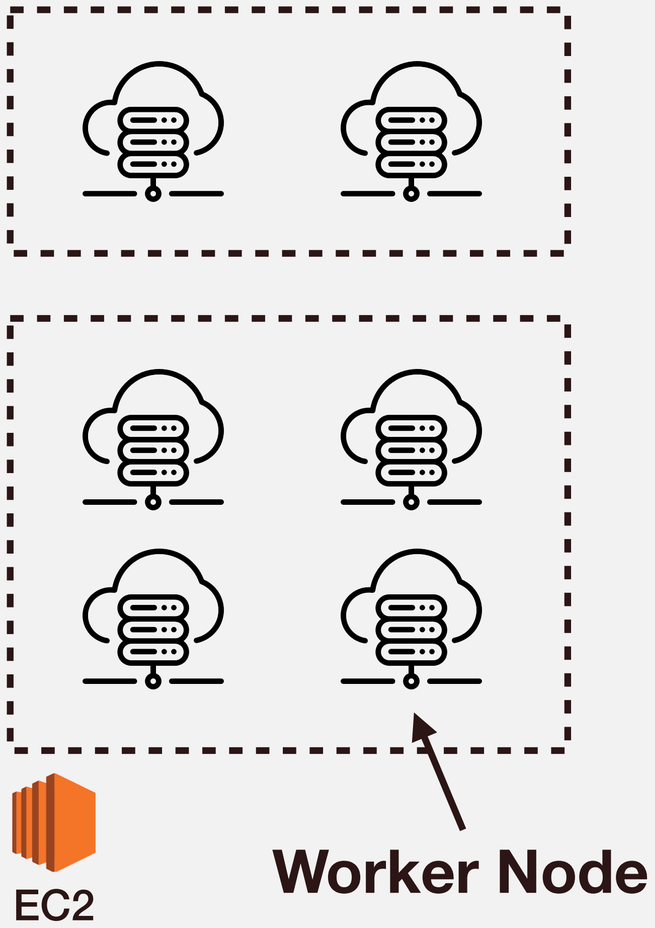
- **Lesson 1**: Dictionary Encoding is the most effective lightweight encoding algorithm for a columnar storage format because most real-world data have low NDV ratios.
- **Lesson 2**: It is important to keep the encoding scheme simple in a columnar storage format to guarantee a competitive scan + decoding performance.
- **Lesson 3**: A columnar storage format should enable block compression cautiously on modern hardware because the bottleneck of query processing is shifting from storage to computation.
- **Lesson 4**: The metadata layout in a columnar storage format should optimize for fewer random probes, especially with cloud storage.

Much better support for ML workloads!

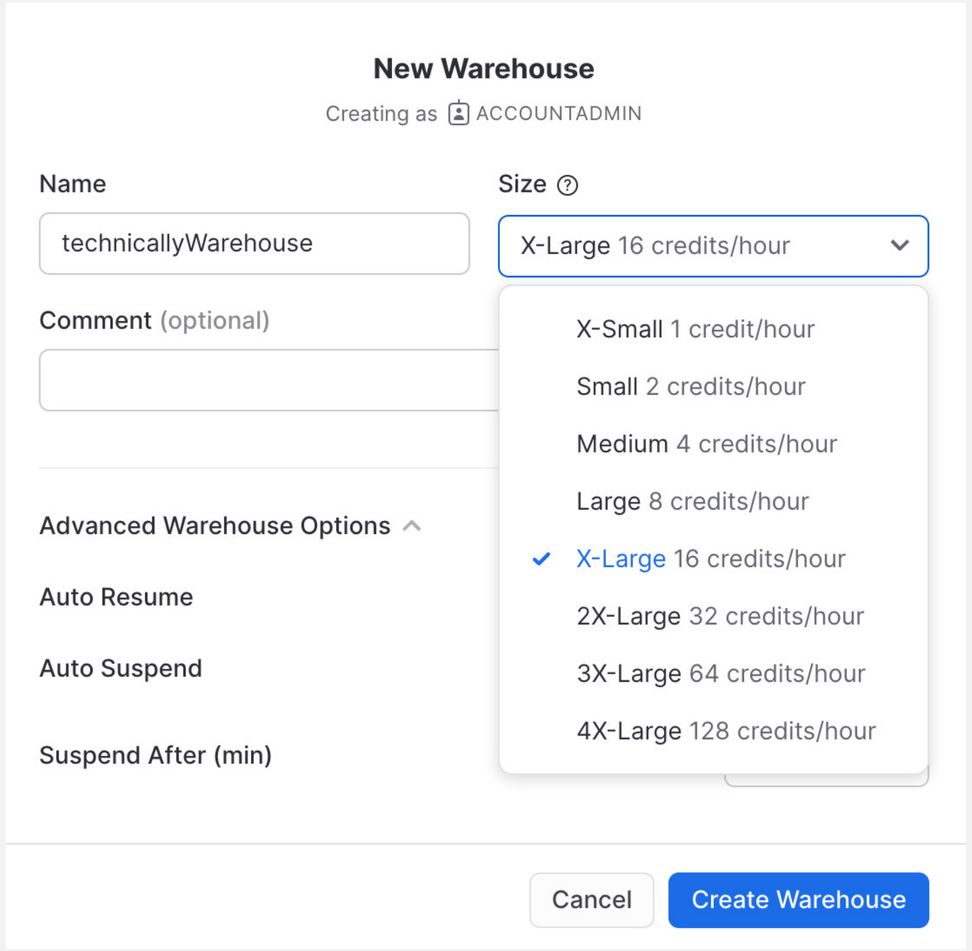
Snowflake Architecture



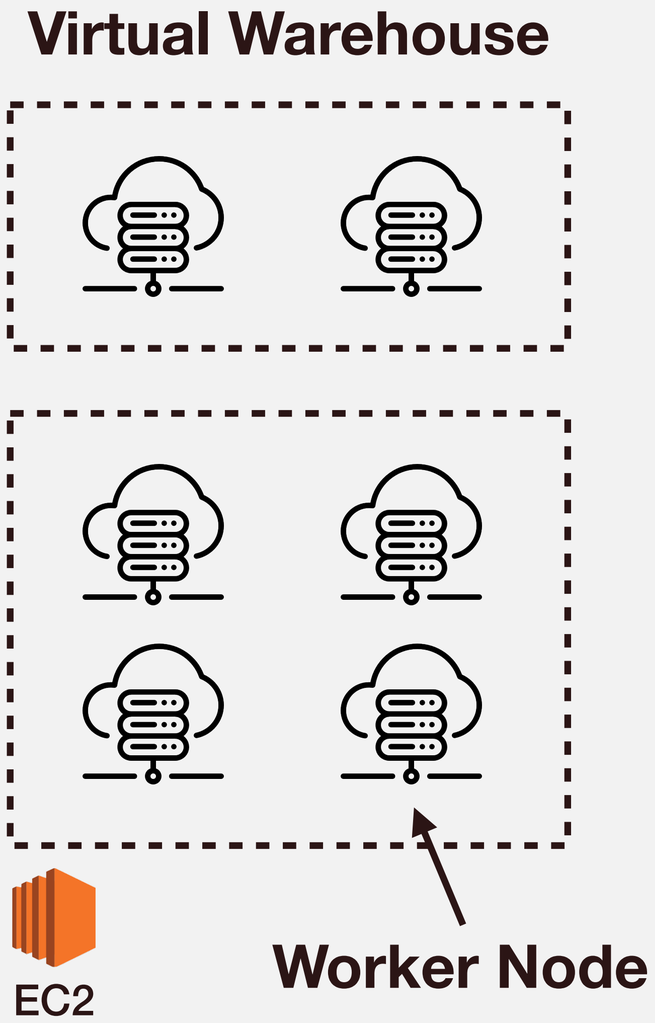
Virtual Warehouses: the Muscle



➜ Create, destroy, resize on demand

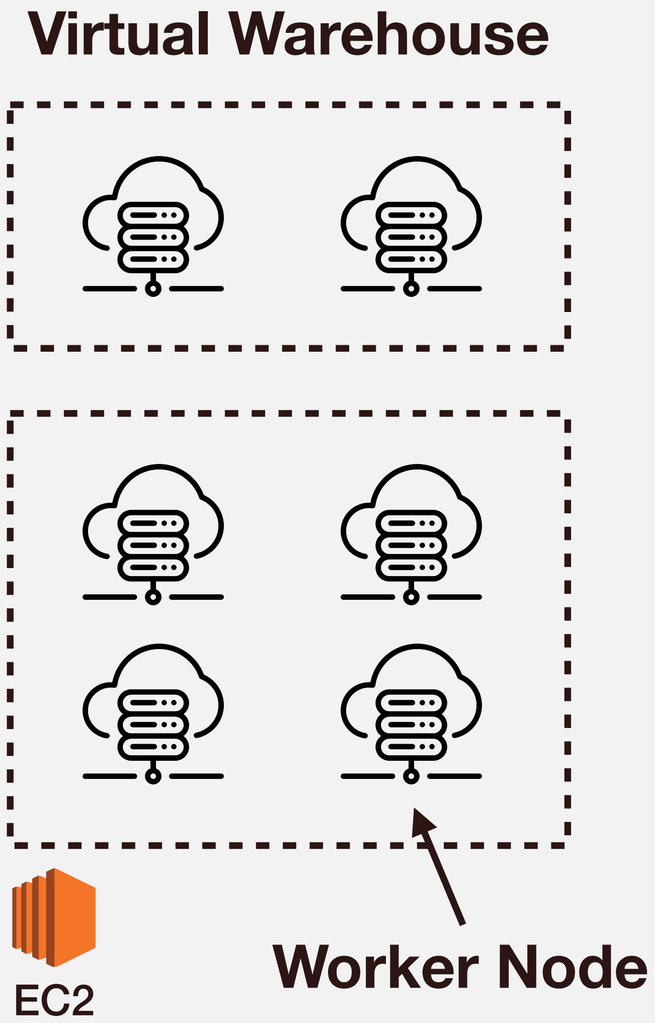


Virtual Warehouses: the Muscle



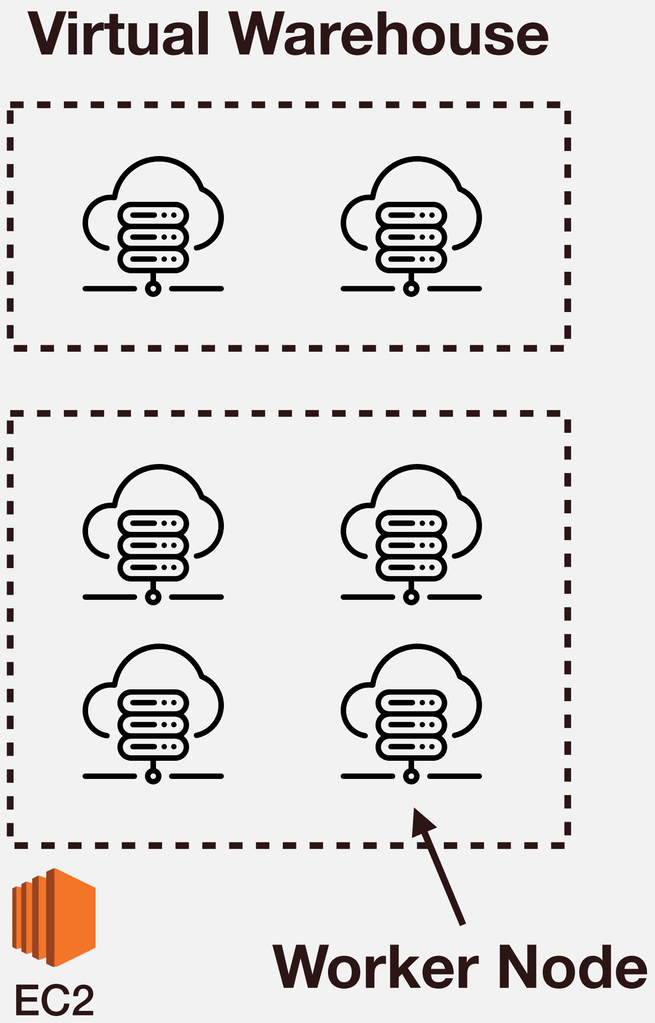
- ➜ Create, destroy, resize on demand
- ➜ Performance Isolation
  - Shared data, private compute

Virtual Warehouses: the Muscle



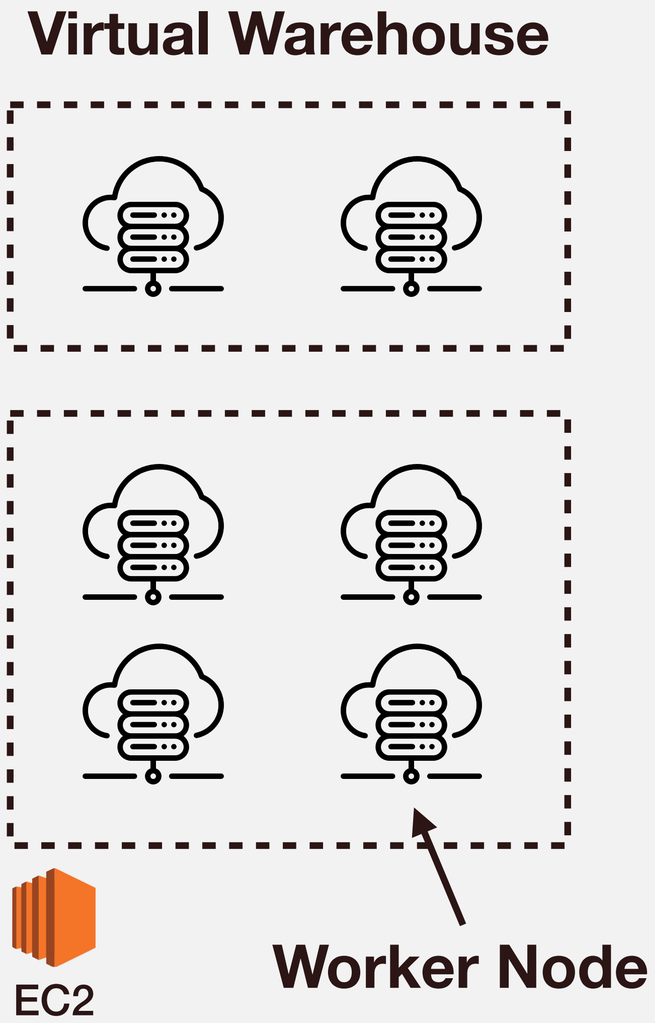
- ➜ Create, destroy, resize on demand
- ➜ Performance Isolation
  - Shared data, private compute
  - Typical usage pattern
    - Continuously-running VWs for repeating jobs
    - On-demand VWs for ad-hoc tasks

Virtual Warehouses: the Muscle



- ➜ Create, destroy, resize on demand
- ➜ Performance Isolation
  - Shared data, private compute
  - Typical usage pattern
    - Continuously-running VWs for repeating jobs
    - On-demand VWs for ad-hoc tasks
- ➜ Ephemeral worker processes

Virtual Warehouses: the Muscle



- ➜ Create, destroy, resize on demand
- ➜ Performance Isolation
  - Shared data, private compute
  - Typical usage pattern
    - Continuously-running VWs for repeating jobs
    - On-demand VWs for ad-hoc tasks
- ➜ Ephemeral worker processes
- ➜ **Columnar, Vectorized, Push-Based**

Execution Engine Design Space

➜ **Engine Type**

- Interpretation
- Compilation (Code-Gen)

➜ **Execution Model**

- Iterator / Volcano
- Fully-Materialized
- Vectorization

➜ **Pipeline Direction**

- Pull
- Push

Execution Engine Design Space

➜ **Engine Type**

- **Interpretation**
- Compilation (Code-Gen)

➜ **Execution Model**

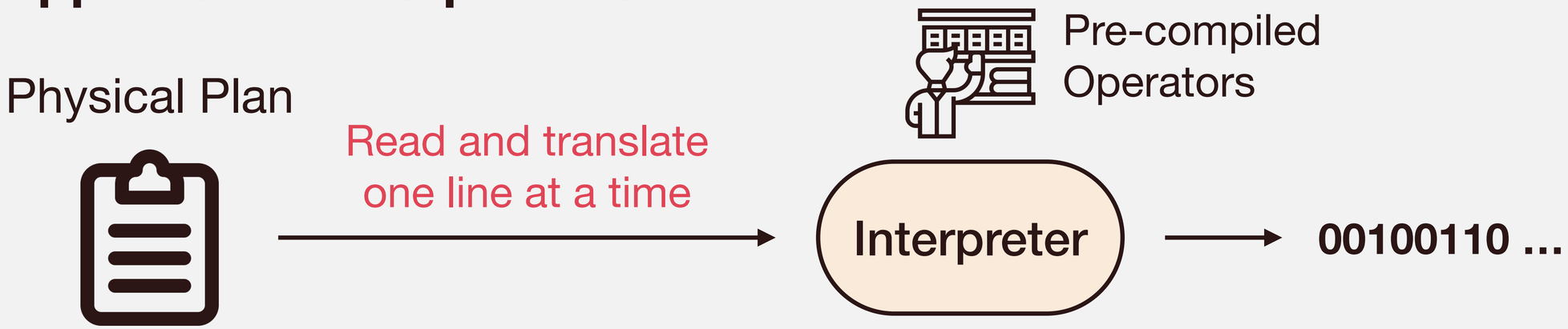
- Iterator / Volcano
- Fully-Materialized
- **Vectorization**

➜ **Pipeline Direction**

- Pull
- **Push**

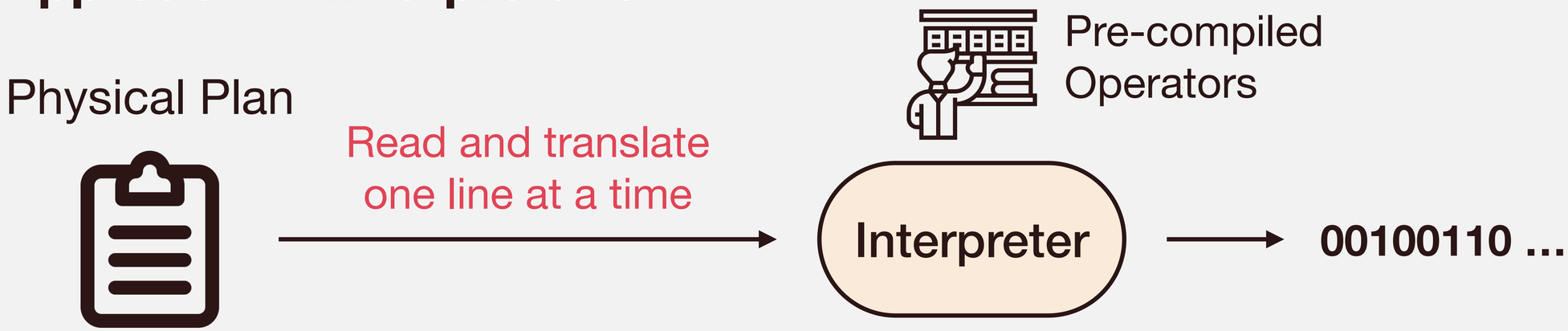
Executing the Plan

➔ **Approach 1: Interpretation**

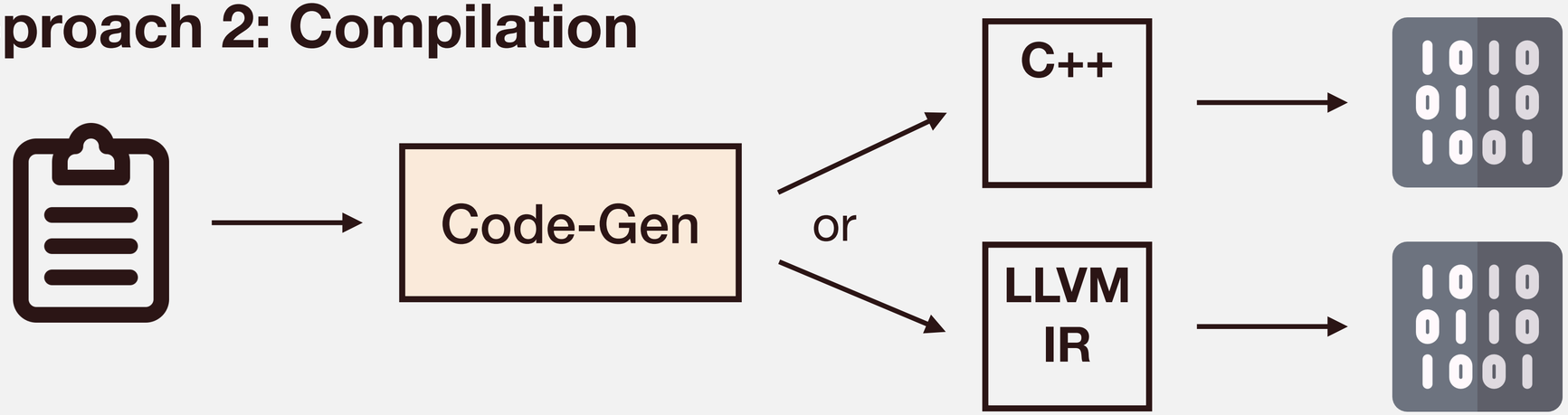


Executing the Plan

➔ **Approach 1: Interpretation**

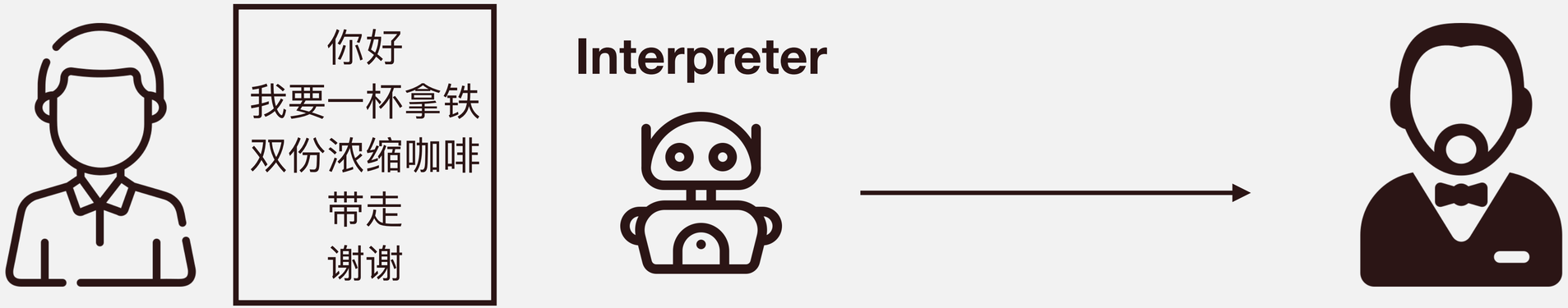


➔ **Approach 2: Compilation**



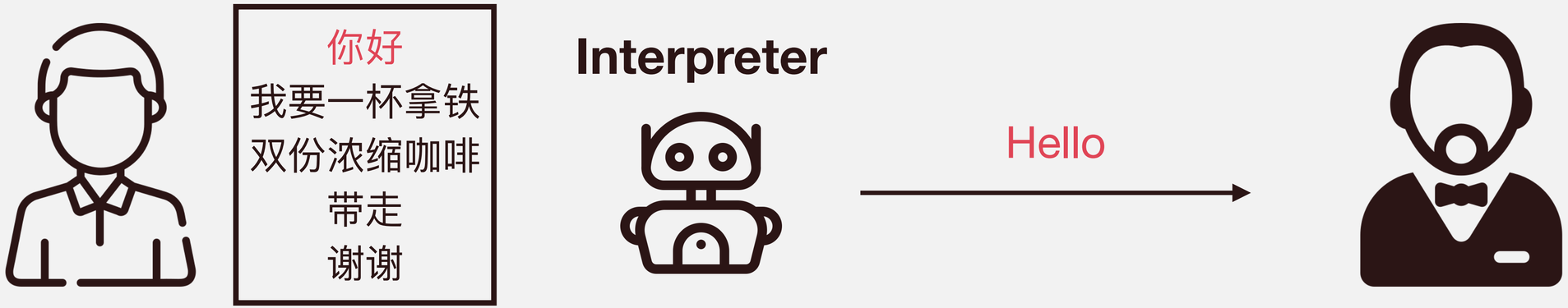
Aside: Interpreting vs. Compiling

➜ **Interpreter**: bring the translator all the time



Aside: Interpreting vs. Compiling

➜ **Interpreter**: bring the translator all the time



Aside: Interpreting vs. Compiling

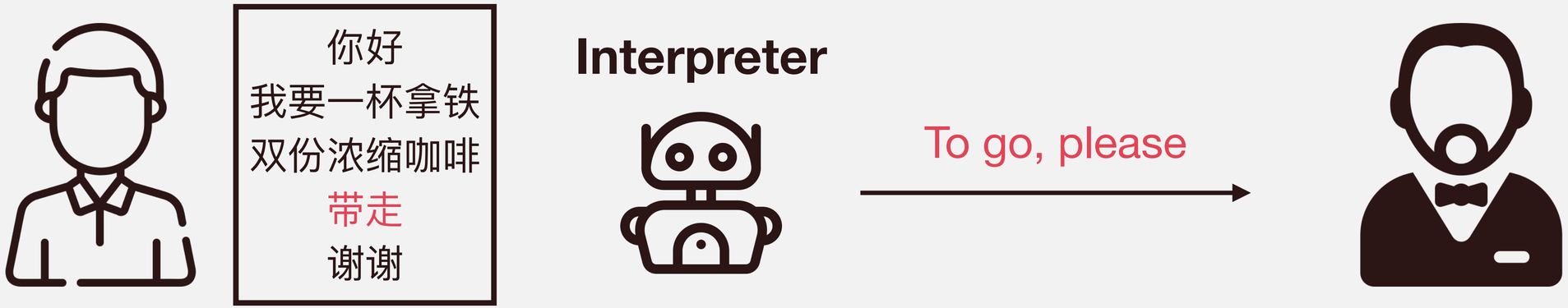
➜ **Interpreter**: bring the translator all the time

Aside: Interpreting vs. Compiling

➜ **Interpreter**: bring the translator all the time

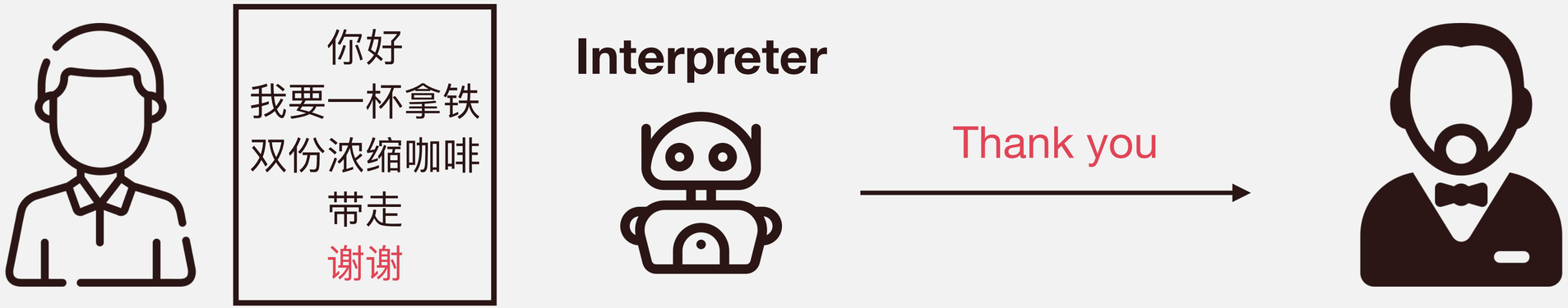
Aside: Interpreting vs. Compiling

➜ **Interpreter**: bring the translator all the time



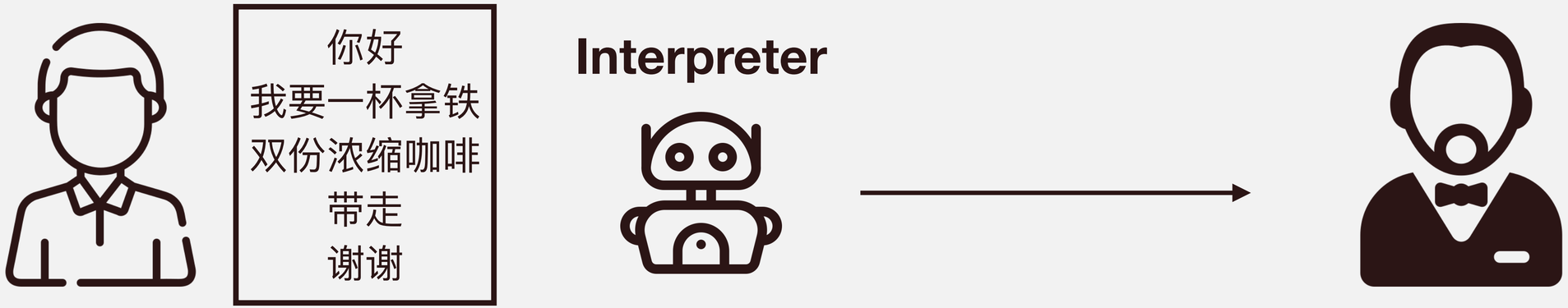
Aside: Interpreting vs. Compiling

➜ **Interpreter**: bring the translator all the time



Aside: Interpreting vs. Compiling

➜ **Interpreter**: bring the translator all the time

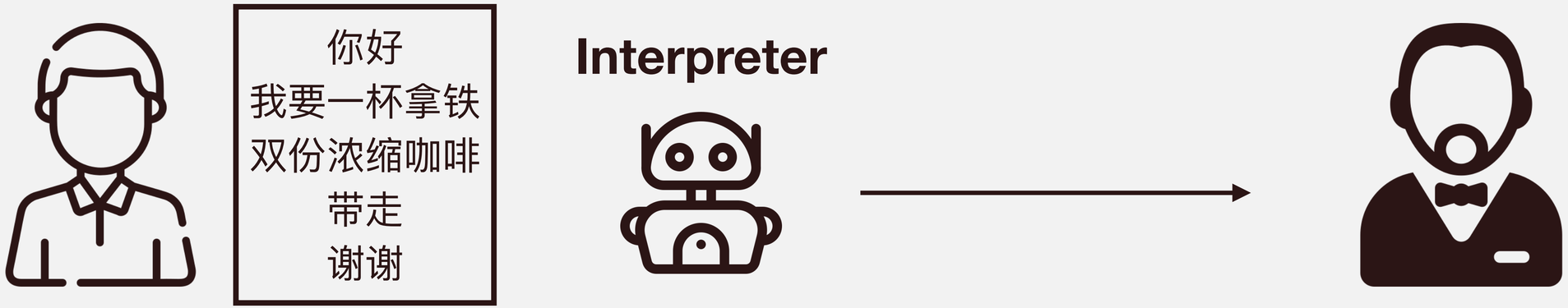


Pros

Cons

Aside: Interpreting vs. Compiling

➜ **Interpreter**: bring the translator all the time



Pros

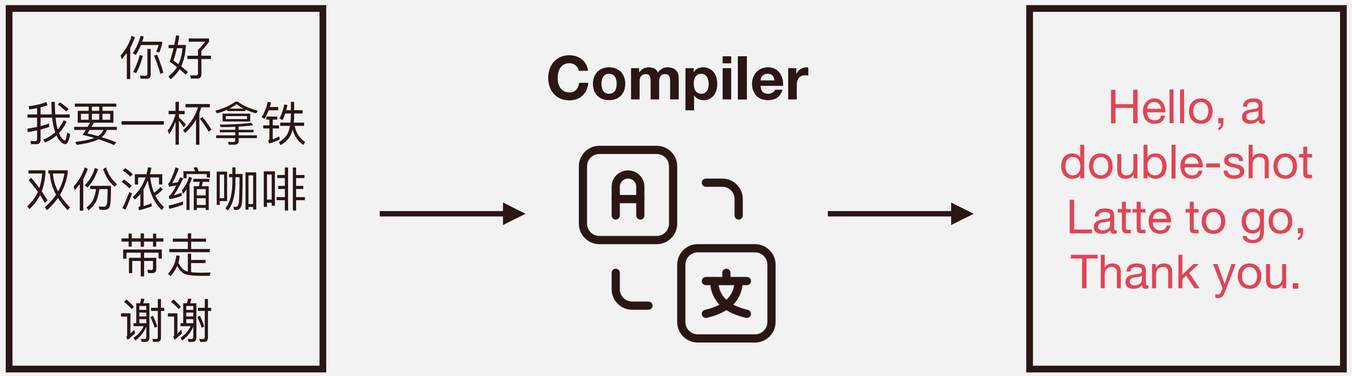
- No need for code analysis beforehand
- Easier to test and debug
- Cross-platform

Cons

- Need interpreters
- Execution is often slower

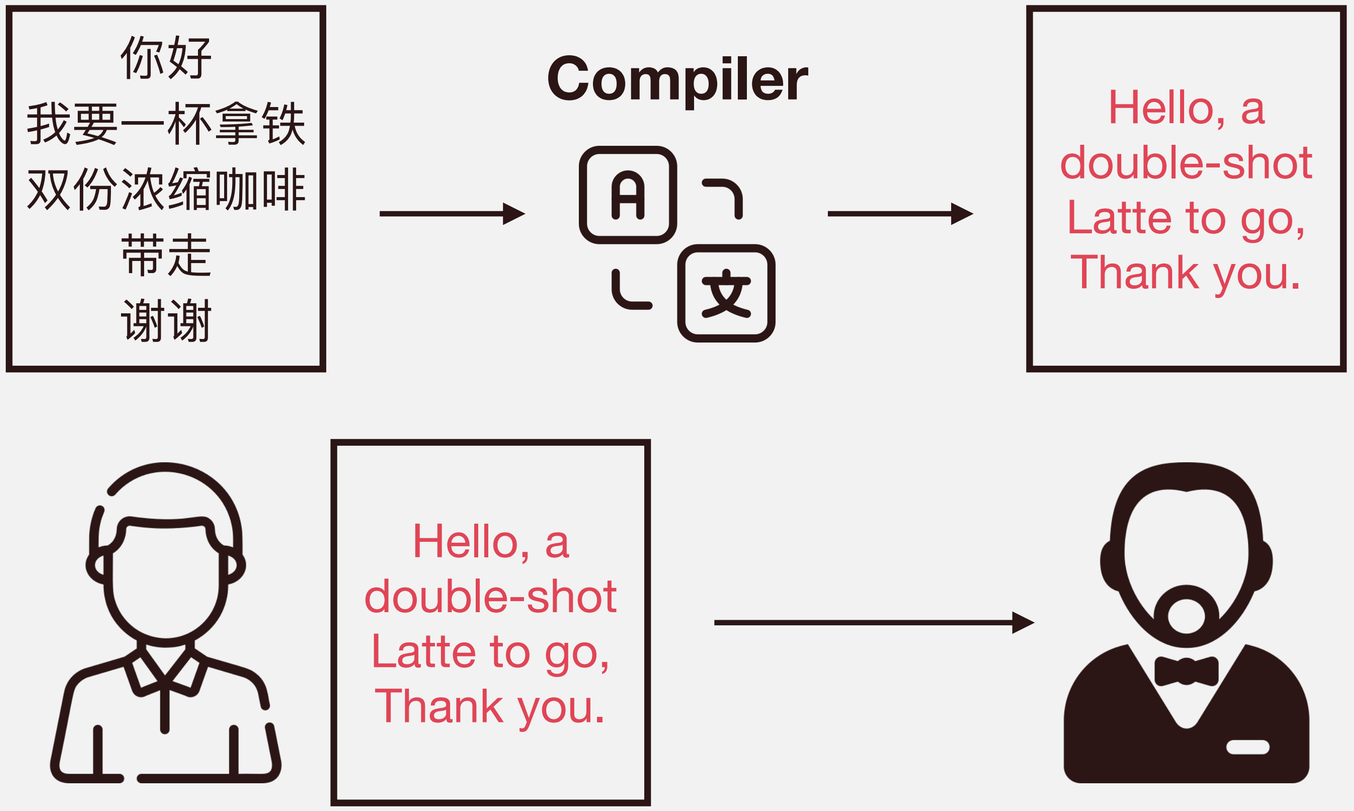
Aside: Interpreting vs. Compiling

➜ **Complier**: bring the pre-translated sheet



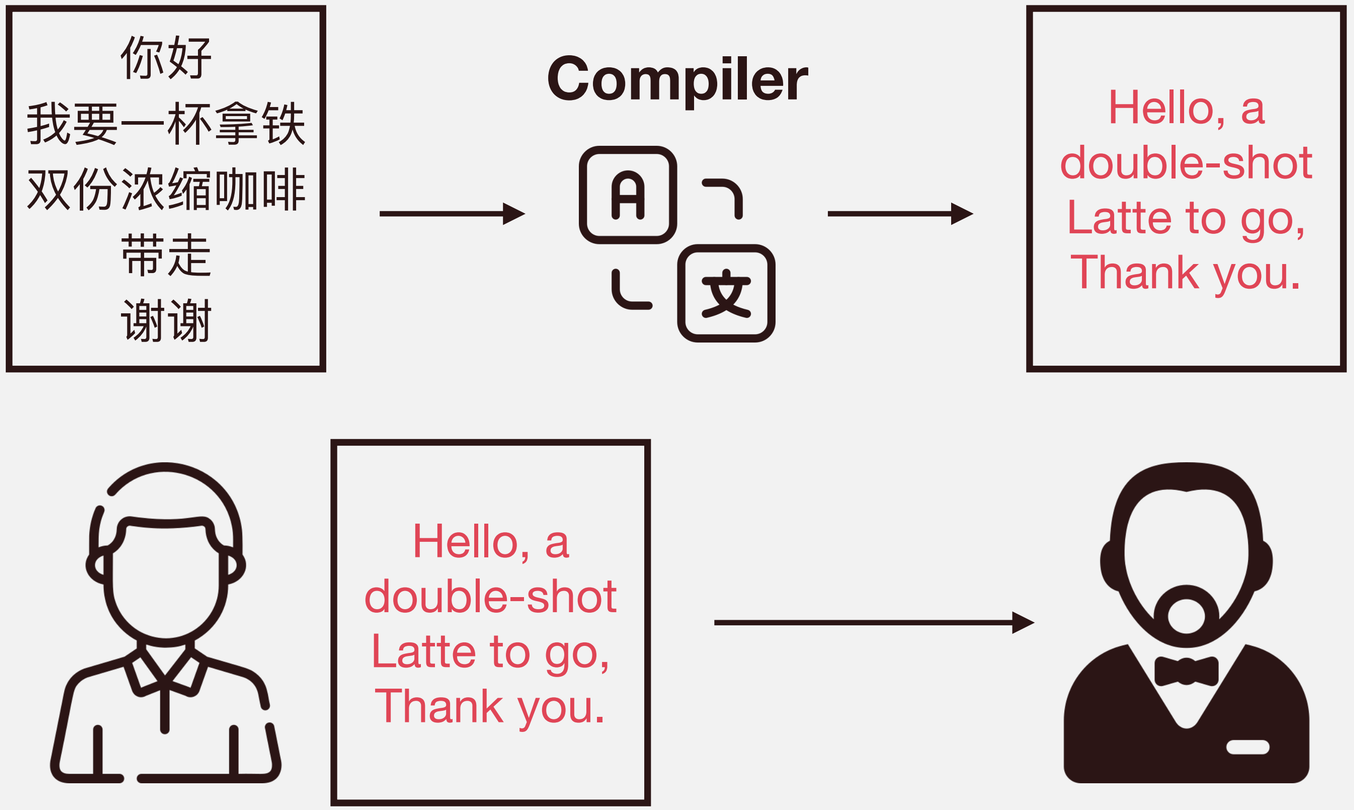
Aside: Interpreting vs. Compiling

➜ **Complier**: bring the pre-translated sheet



Aside: Interpreting vs. Compiling

➜ **Complier**: bring the pre-translated sheet



Pros

- Faster, ready-to-run
- Code better optimized

Cons

- Long extra compile time
- Requires more memory
- Hard to get it right
- Worse portability

Execution Engine Design Space

➜ **Engine Type**

- **Interpretation**
- Compilation (Code-Gen)

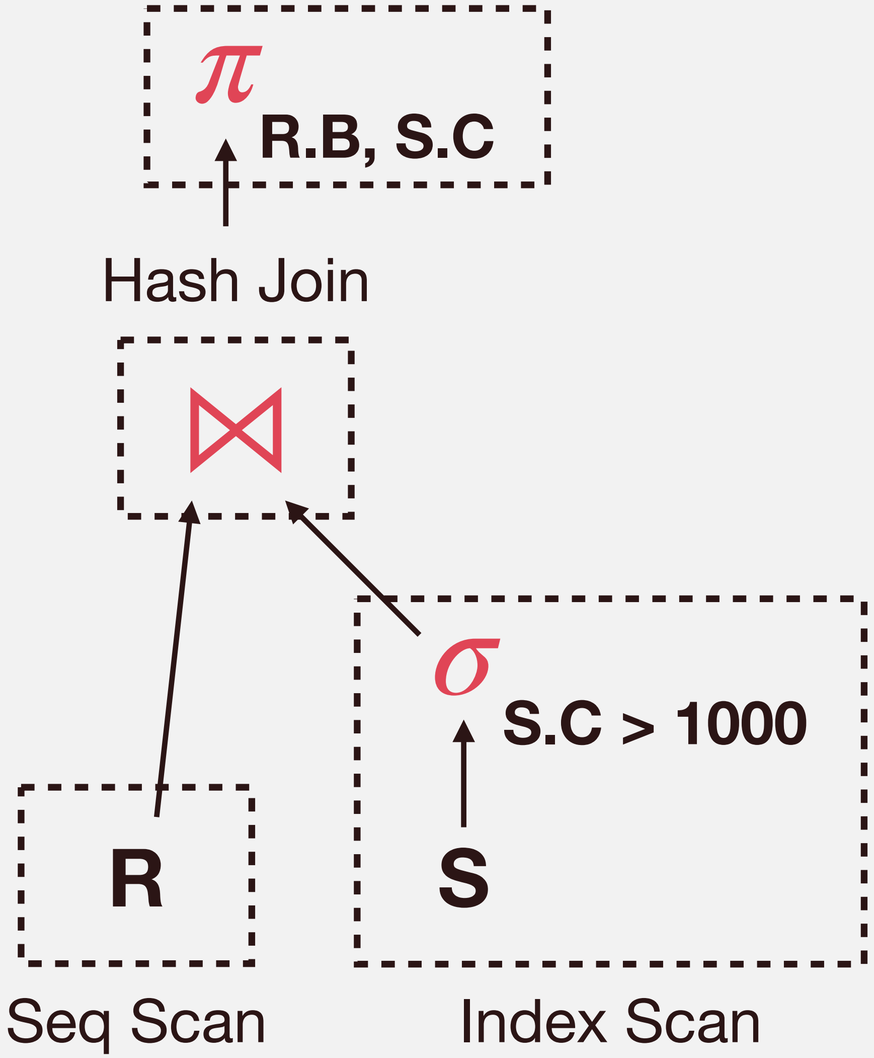
➜ **Execution Model**

- Iterator / Volcano
- Fully-Materialized
- **Vectorization**

➜ **Pipeline Direction**

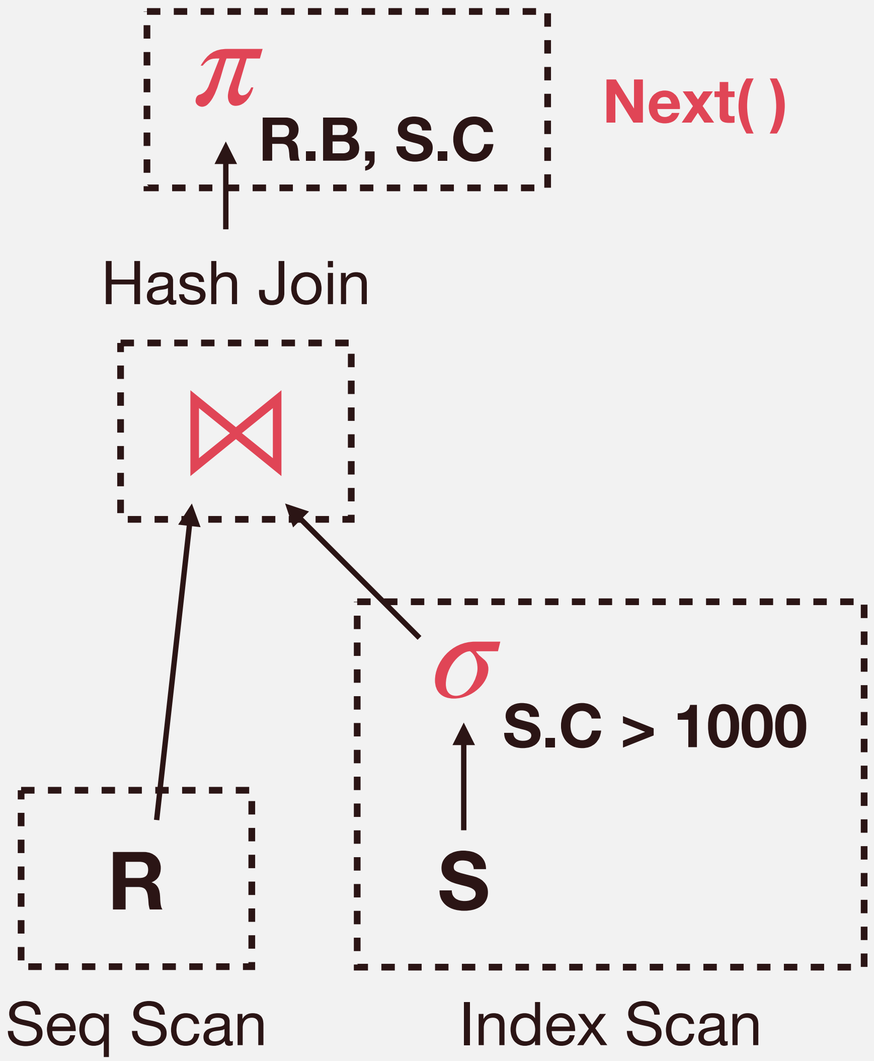
- Pull
- **Push**

Iterator/Volcano Model



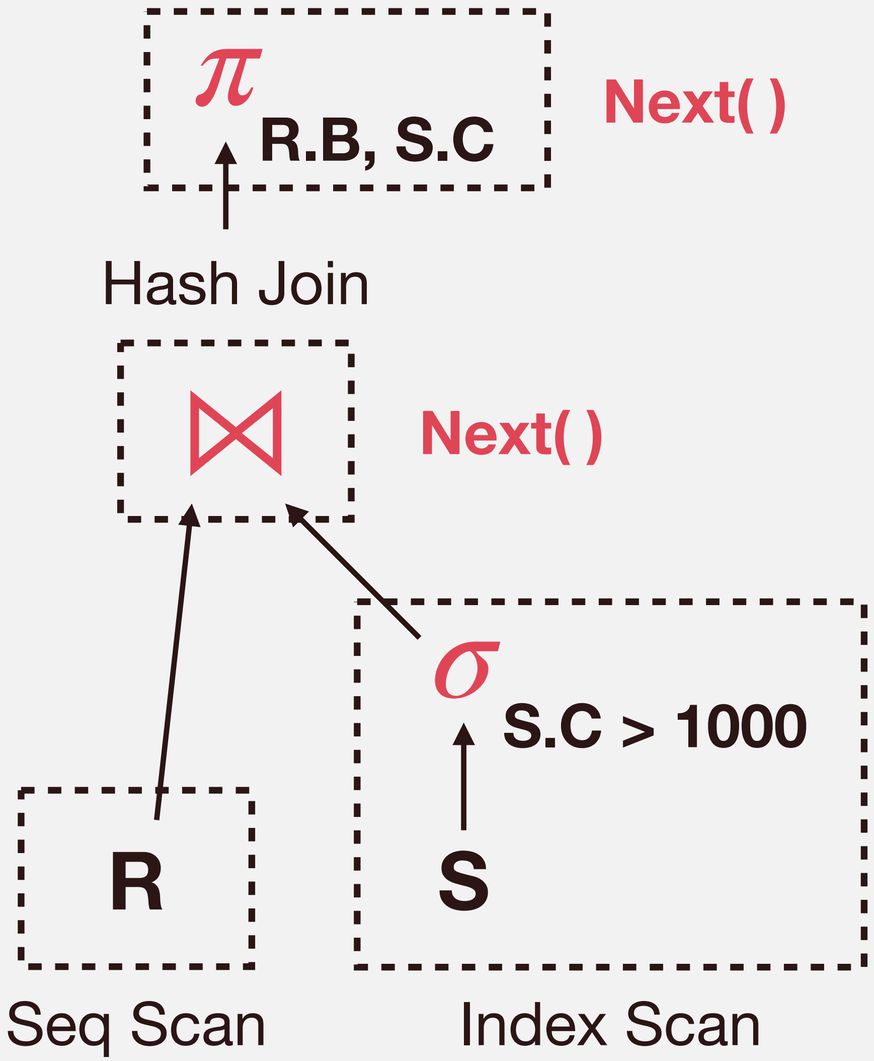
- ➜ Every operator implements **Next( )**
  - Emits an output tuple or NULL
- ➜ The root operator implements a **loop** that keeps invoking **Next( )** on its child
- ➜ Execution can be **pipelined**
- ➜ Could have **pipeline breakers**
  - E.g., join, order by
- ➜ Elegant implementation

Iterator/Volcano Model



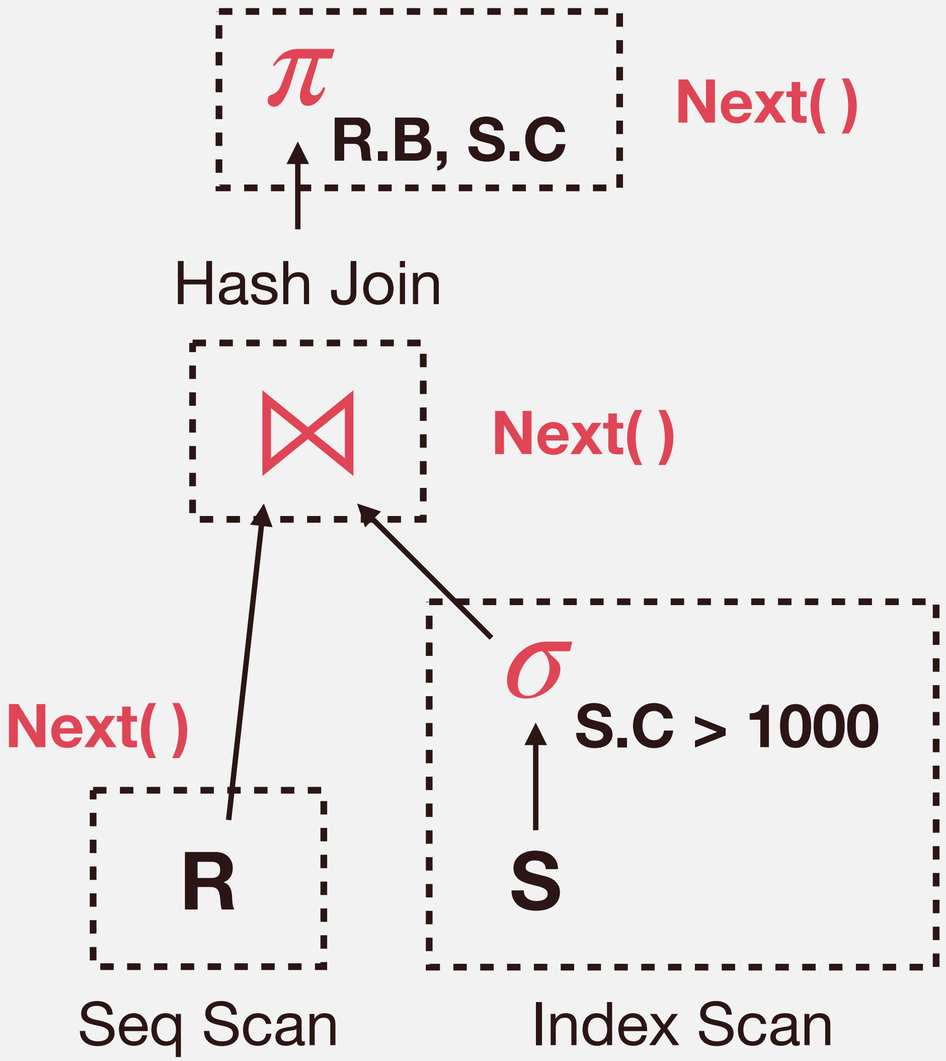
- ➜ Every operator implements **Next( )**
  - Emits an output tuple or NULL
- ➜ The root operator implements a **loop** that keeps invoking **Next( )** on its child
- ➜ Execution can be **pipelined**
- ➜ Could have **pipeline breakers**
  - E.g., join, order by
- ➜ Elegant implementation

Iterator/Volcano Model



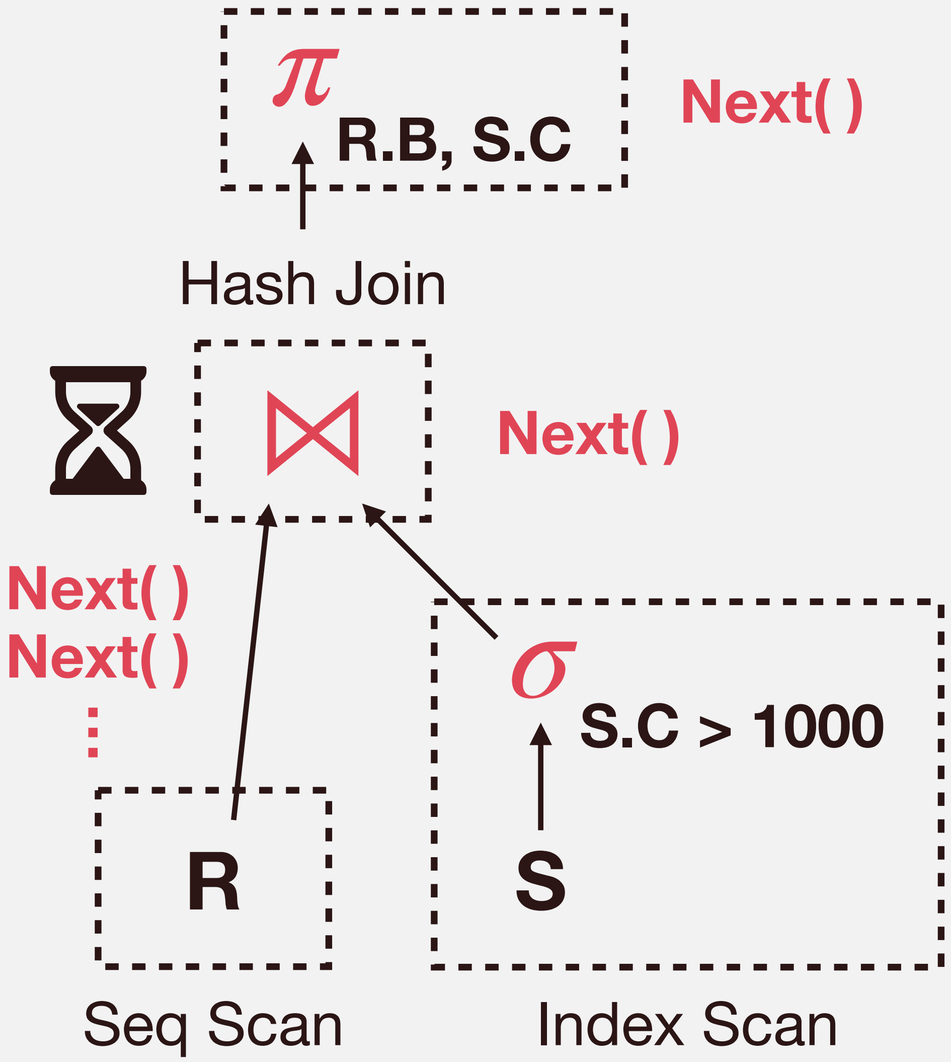
- ➜ Every operator implements **Next( )**
  - Emits an output tuple or NULL
- ➜ The root operator implements a **loop** that keeps invoking **Next( )** on its child
- ➜ Execution can be **pipelined**
- ➜ Could have **pipeline breakers**
  - E.g., join, order by
- ➜ Elegant implementation

Iterator/Volcano Model



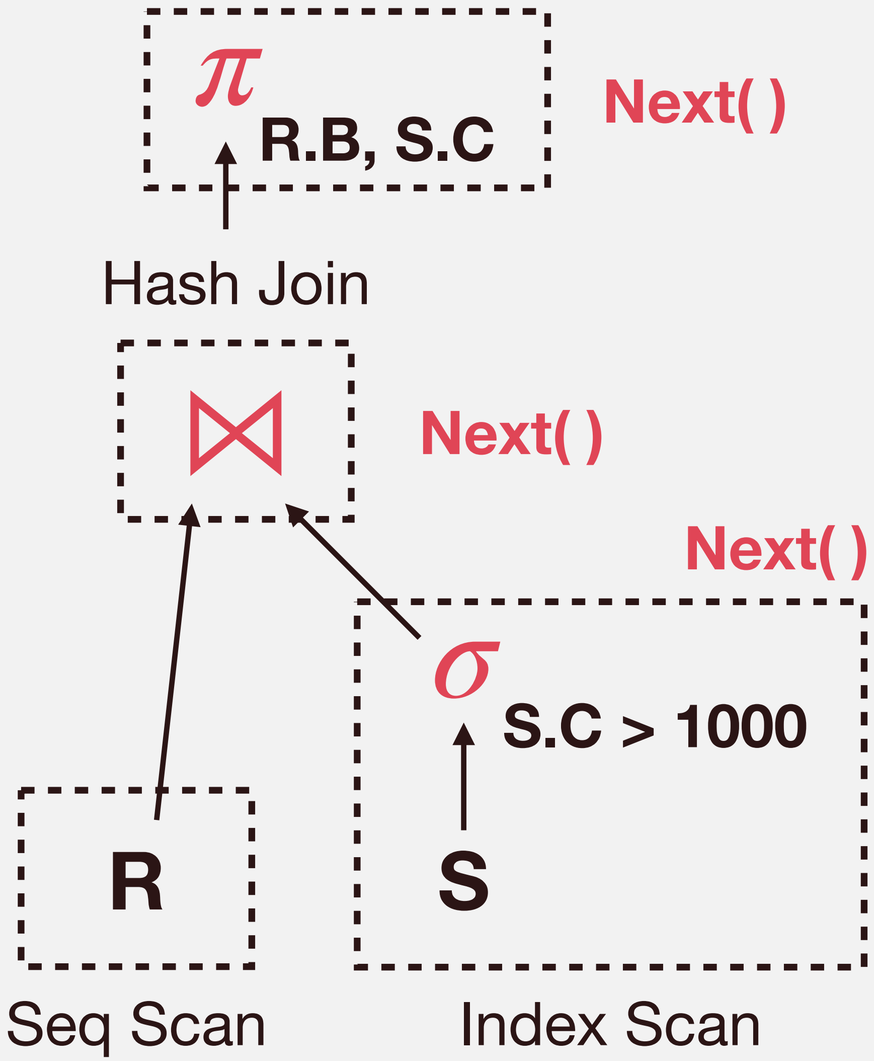
- ➜ Every operator implements **Next( )**
  - Emits an output tuple or NULL
- ➜ The root operator implements a **loop** that keeps invoking **Next( )** on its child
- ➜ Execution can be **pipelined**
- ➜ Could have **pipeline breakers**
  - E.g., join, order by
- ➜ Elegant implementation

Iterator/Volcano Model



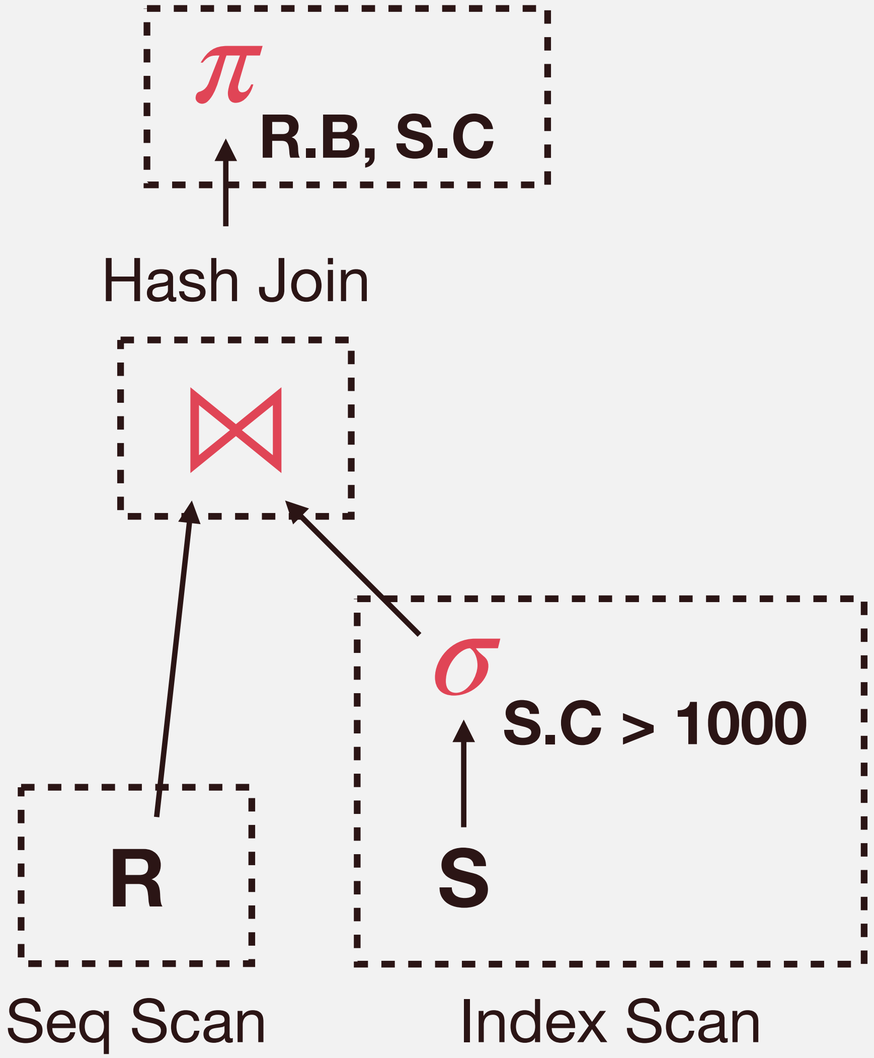
- ➜ Every operator implements **Next( )**
  - Emits an output tuple or NULL
- ➜ The root operator implements a **loop** that keeps invoking **Next( )** on its child
- ➜ Execution can be **pipelined**
- ➜ Could have **pipeline breakers**
  - E.g., join, order by
- ➜ Elegant implementation

Iterator/Volcano Model



- ➜ Every operator implements **Next( )**
  - Emits an output tuple or NULL
- ➜ The root operator implements a **loop** that keeps invoking **Next( )** on its child
- ➜ Execution can be **pipelined**
- ➜ Could have **pipeline breakers**
  - E.g., join, order by
- ➜ Elegant implementation

Iterator/Volcano Model

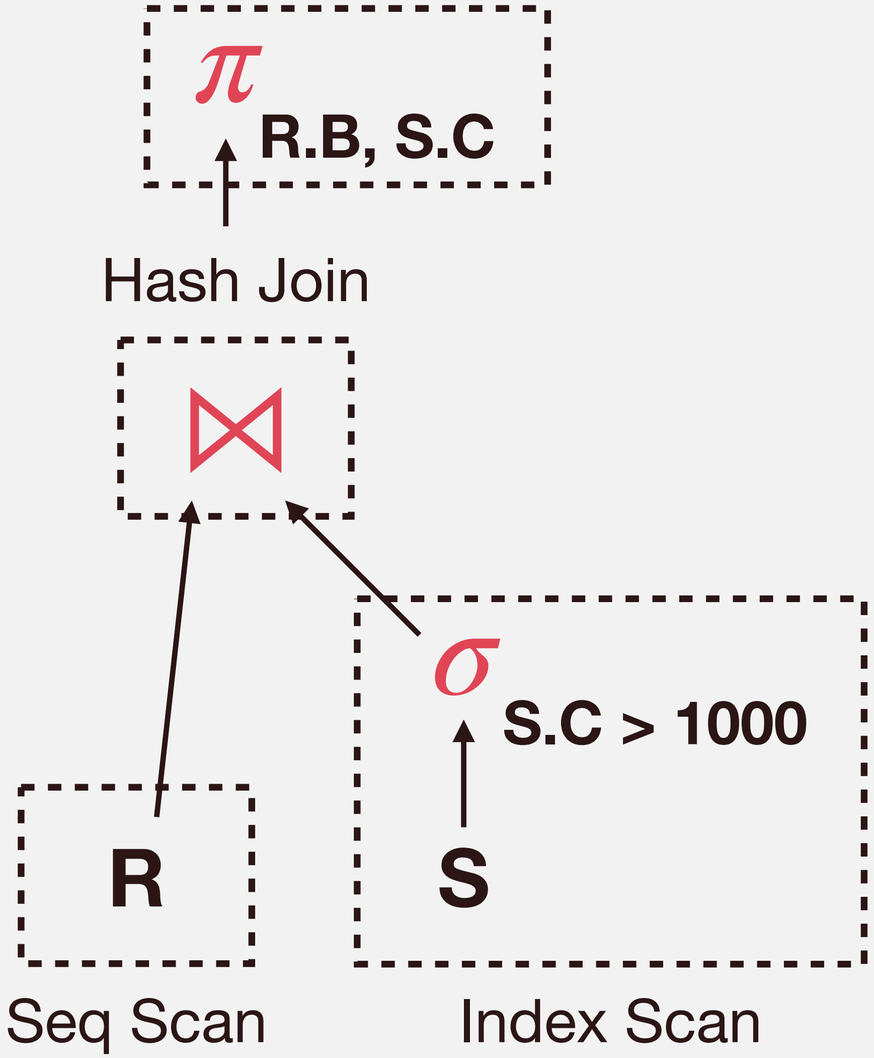



```
class AbstractExecutor {
    virtual void Init() = 0;
    virtual Tuple* Next() = 0;
protected:
    Context *ctx;
};
```

➜ Every operator implements the same interface

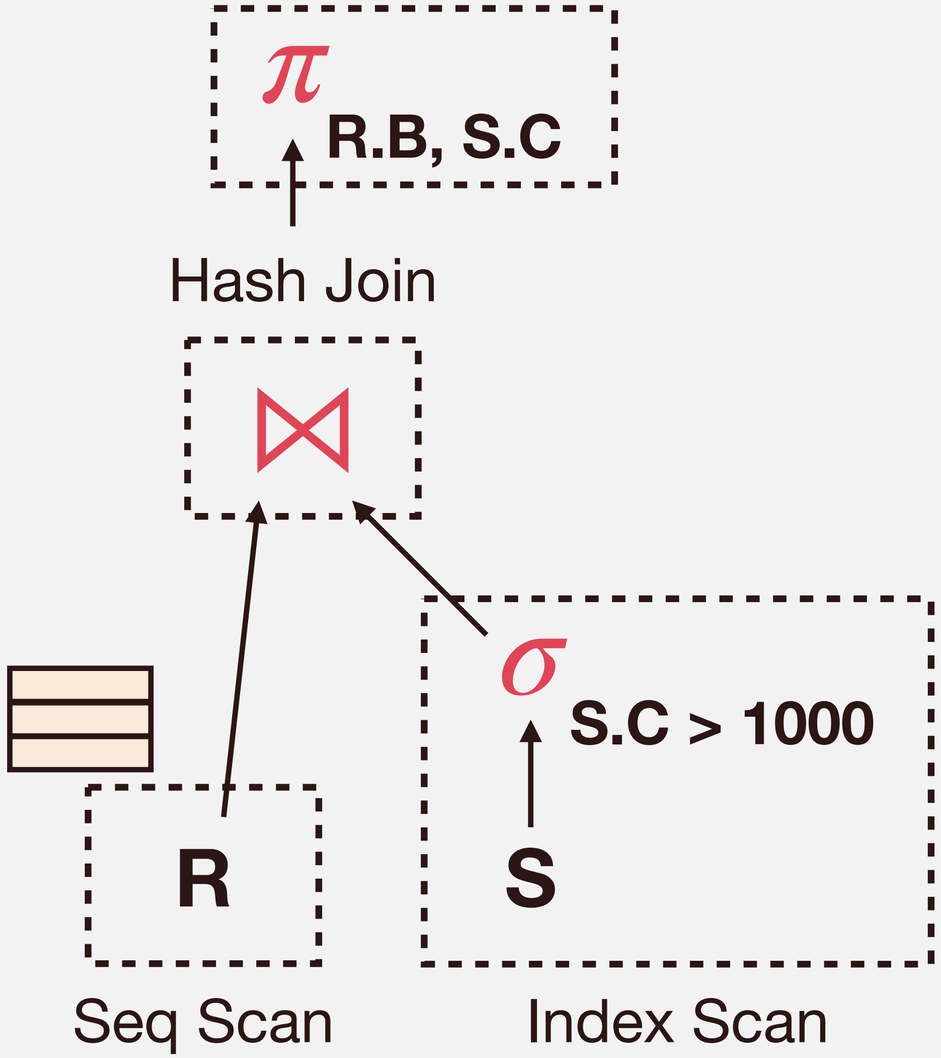
➜ Operators may have internal states

Fully-Materialized Model



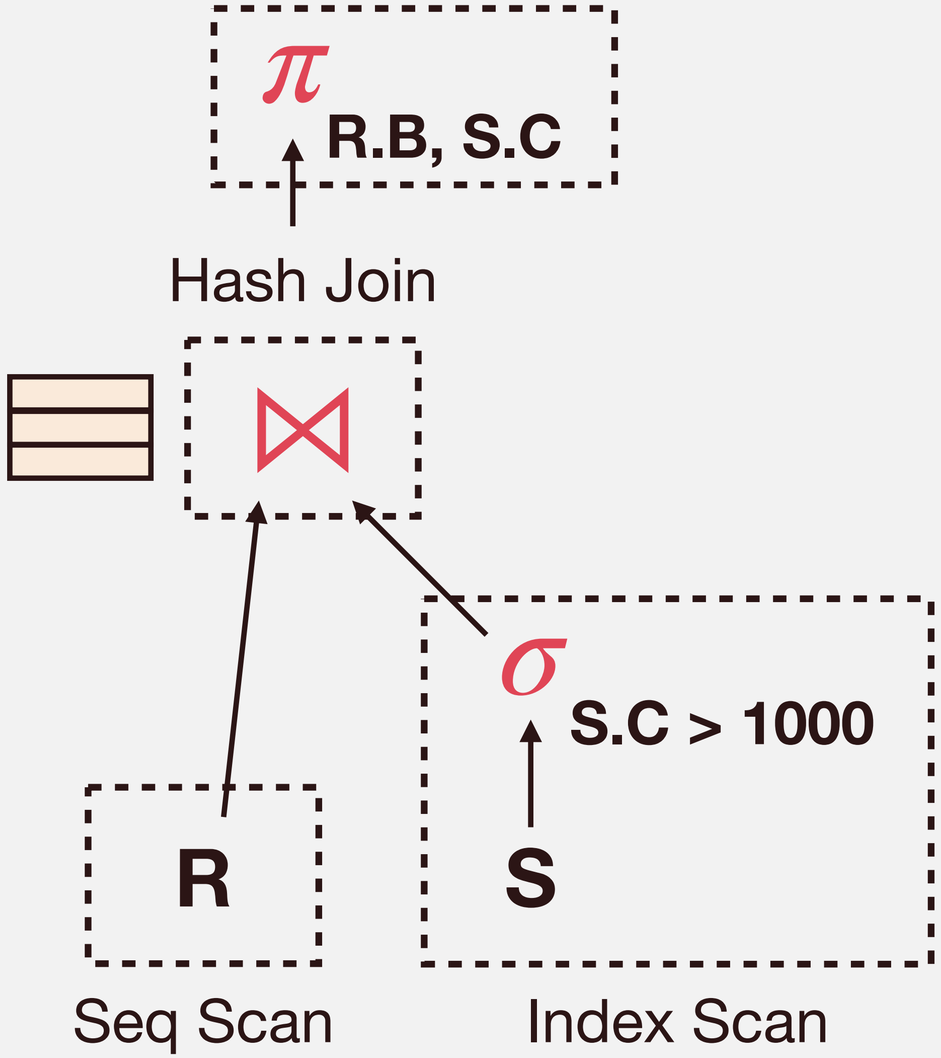
- ➜ Each operator stores its output in a single buffer and returns all at once
- ➜ Parent operator does not start until its children finish
- ➜ Non-pipelined

Fully-Materialized Model



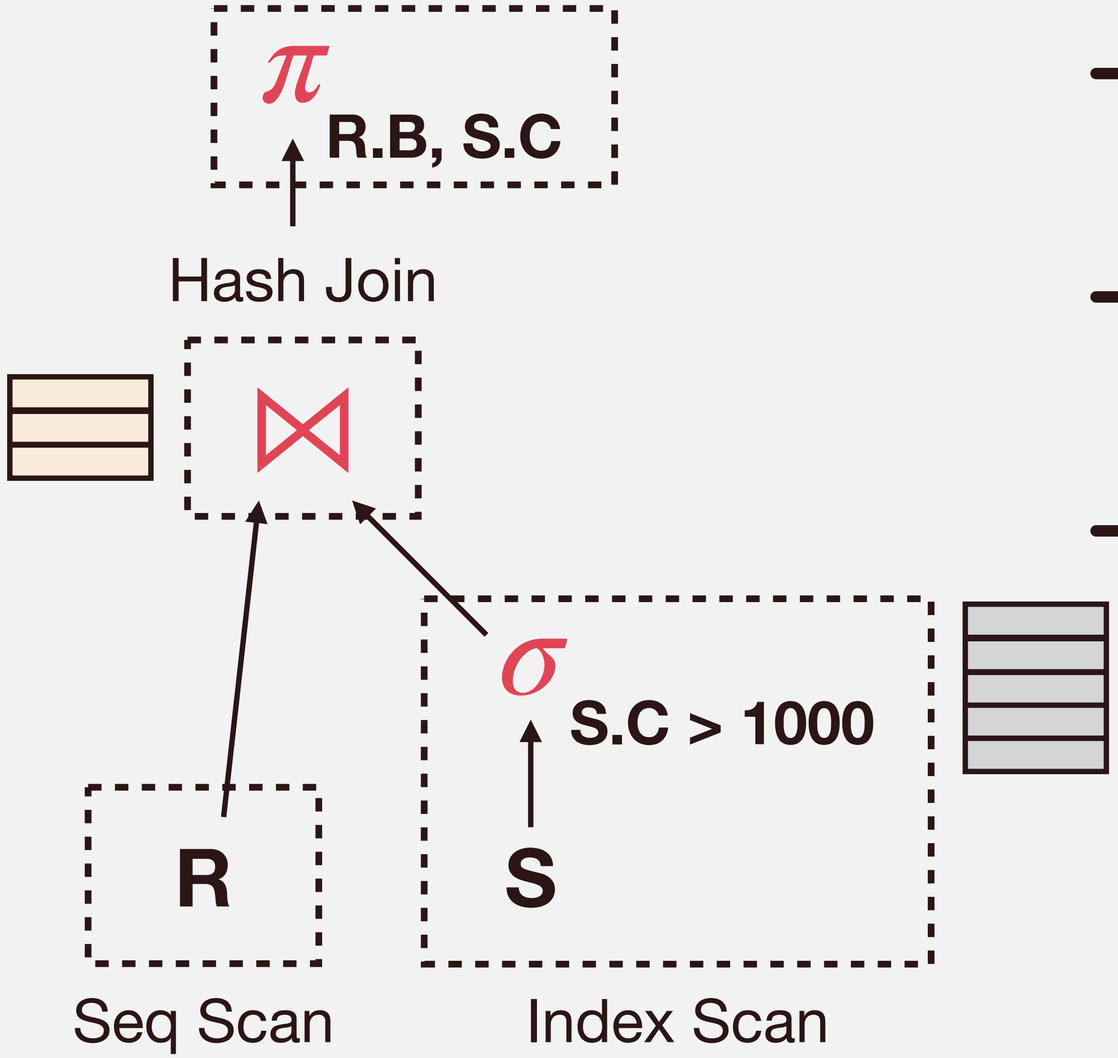
- ➜ Each operator stores its output in a single buffer and returns all at once
- ➜ Parent operator does not start until its children finish
- ➜ Non-pipelined

Fully-Materialized Model



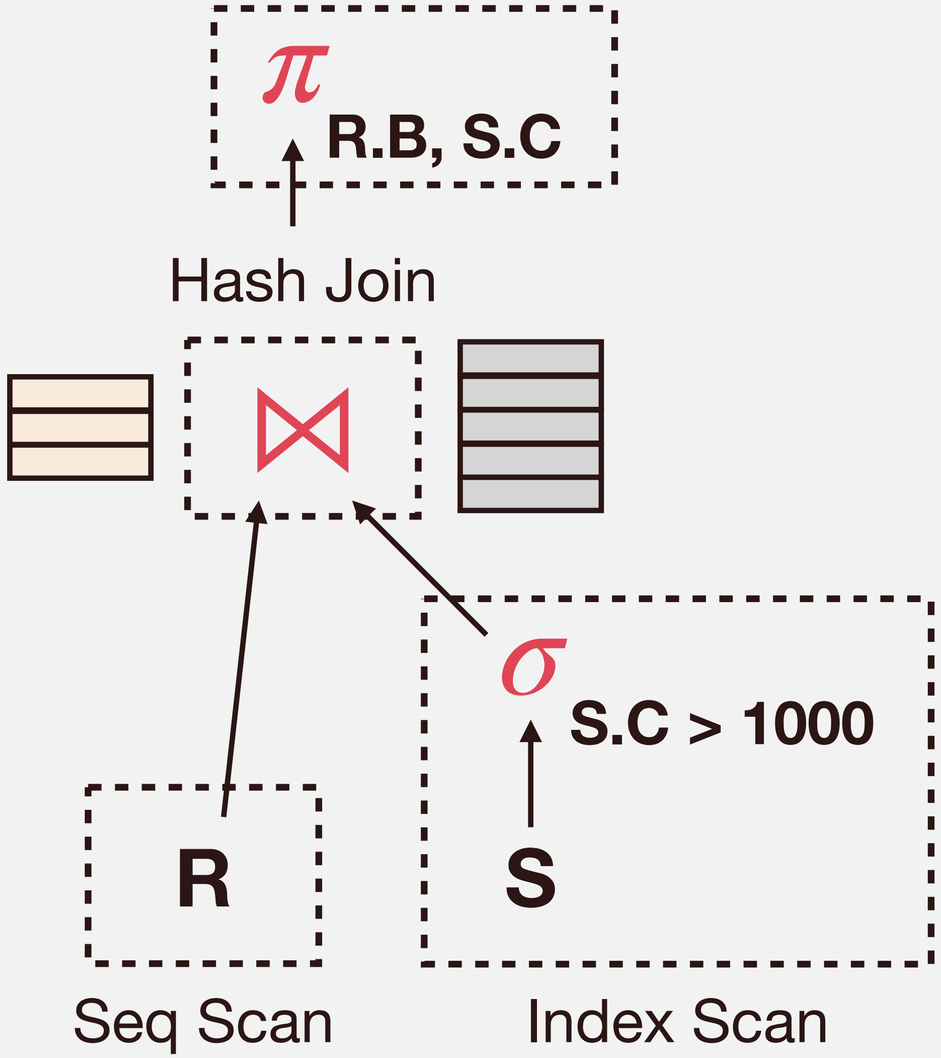
- ➜ Each operator stores its output in a single buffer and returns all at once
- ➜ Parent operator does not start until its children finish
- ➜ Non-pipelined

Fully-Materialized Model



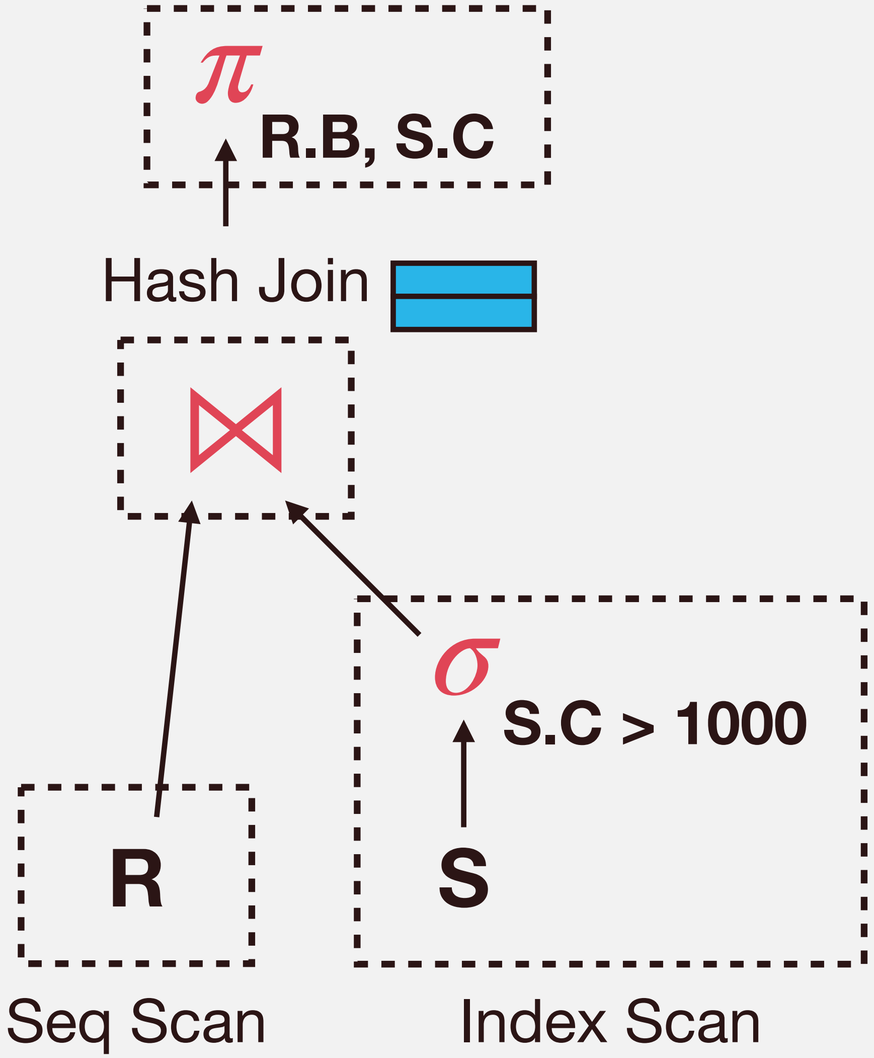
- ➜ Each operator stores its output in a single buffer and returns all at once
- ➜ Parent operator does not start until its children finish
- ➜ Non-pipelined

Fully-Materialized Model



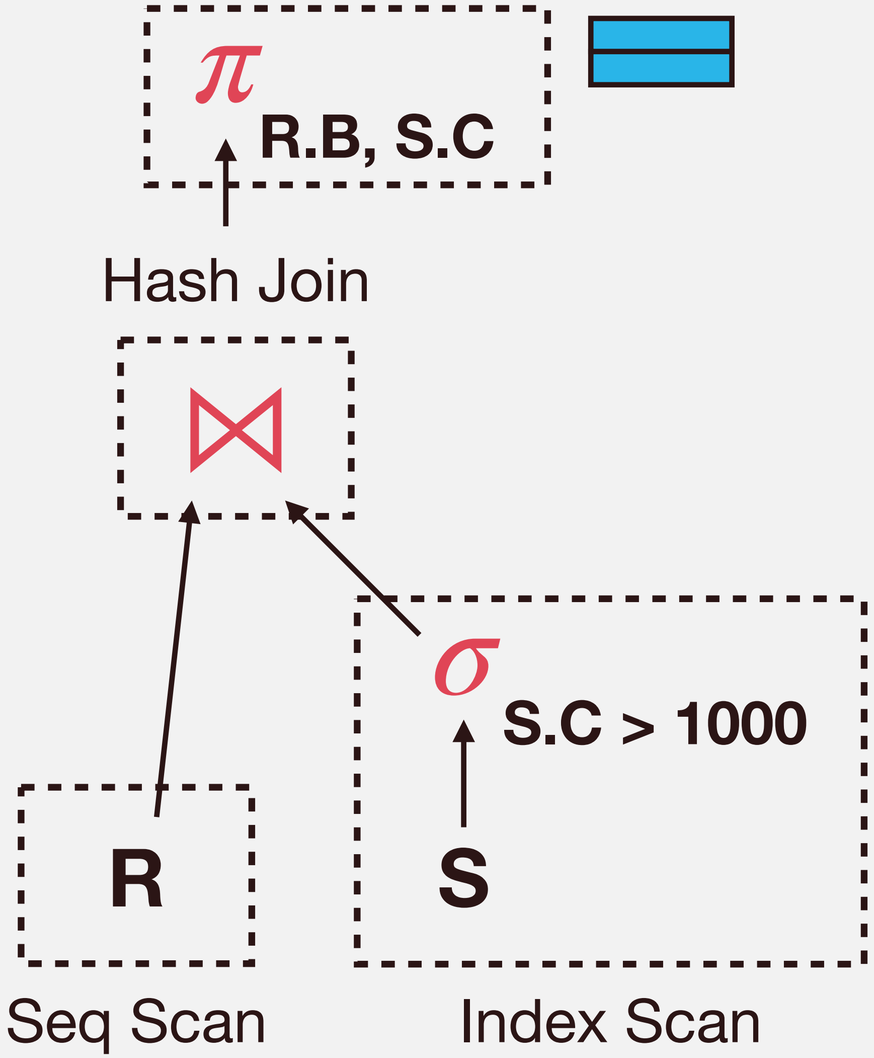
- ➜ Each operator stores its output in a single buffer and returns all at once
- ➜ Parent operator does not start until its children finish
- ➜ Non-pipelined

Fully-Materialized Model



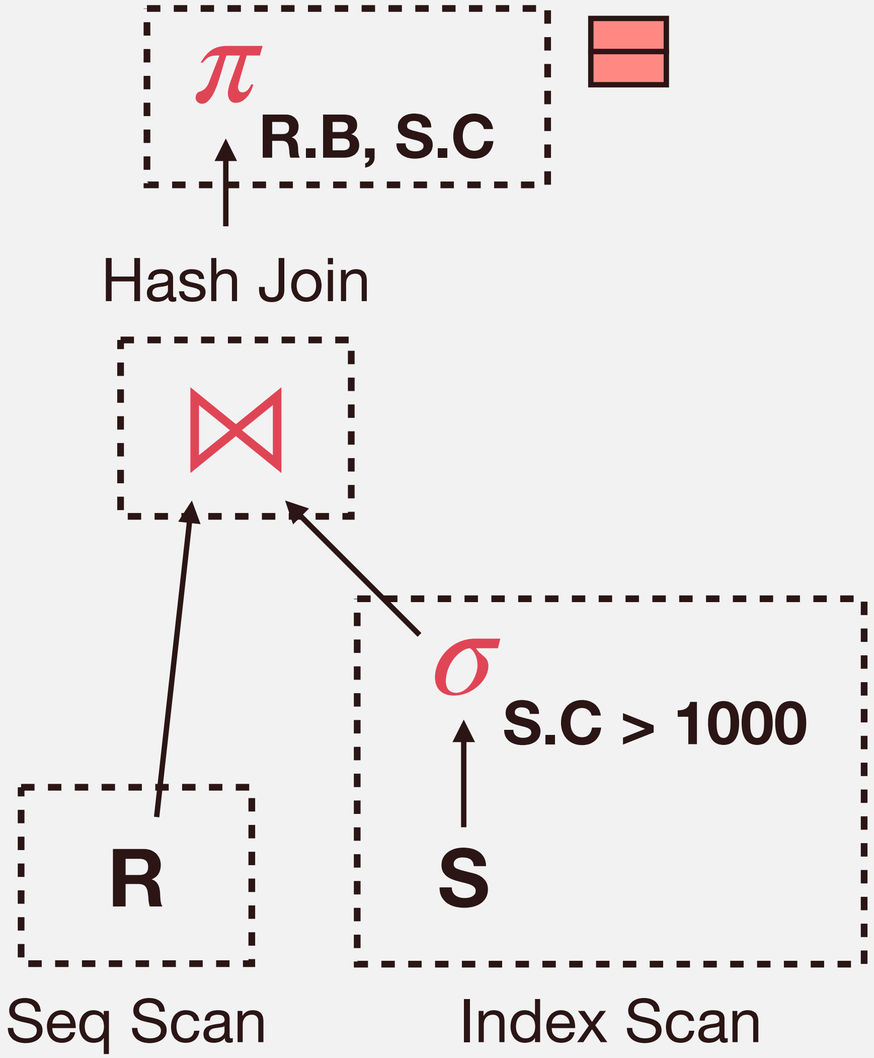
- ➜ Each operator stores its output in a single buffer and returns all at once
- ➜ Parent operator does not start until its children finish
- ➜ Non-pipelined

Fully-Materialized Model



- ➜ Each operator stores its output in a single buffer and returns all at once
- ➜ Parent operator does not start until its children finish
- ➜ Non-pipelined

Fully-Materialized Model



- ➜ Each operator stores its output in a single buffer and returns all at once
- ➜ Parent operator does not start until its children finish
- ➜ Non-pipelined

Iterator vs. Fully-Materialized

Iterator / Volcano

Tuple at a time

Fully-Materialized

Operator at a time

Iterator vs. Fully-Materialized

Iterator / Volcano	Fully-Materialized
Tuple at a time	Operator at a time
Small intermediate results	Need extra memory for intermediate results

Iterator vs. Fully-Materialized

Iterator / Volcano	Fully-Materialized
Tuple at a time	Operator at a time
Small intermediate results	Need extra memory for intermediate results
A lot of virtual function calls	Fewer function calls

Iterator vs. Fully-Materialized

Iterator / Volcano	Fully-Materialized
Tuple at a time	Operator at a time
Small intermediate results	Need extra memory for intermediate results
A lot of virtual function calls	Fewer function calls
Can benefit from pipelining	Can benefit from batch processing (e.g, SIMD)

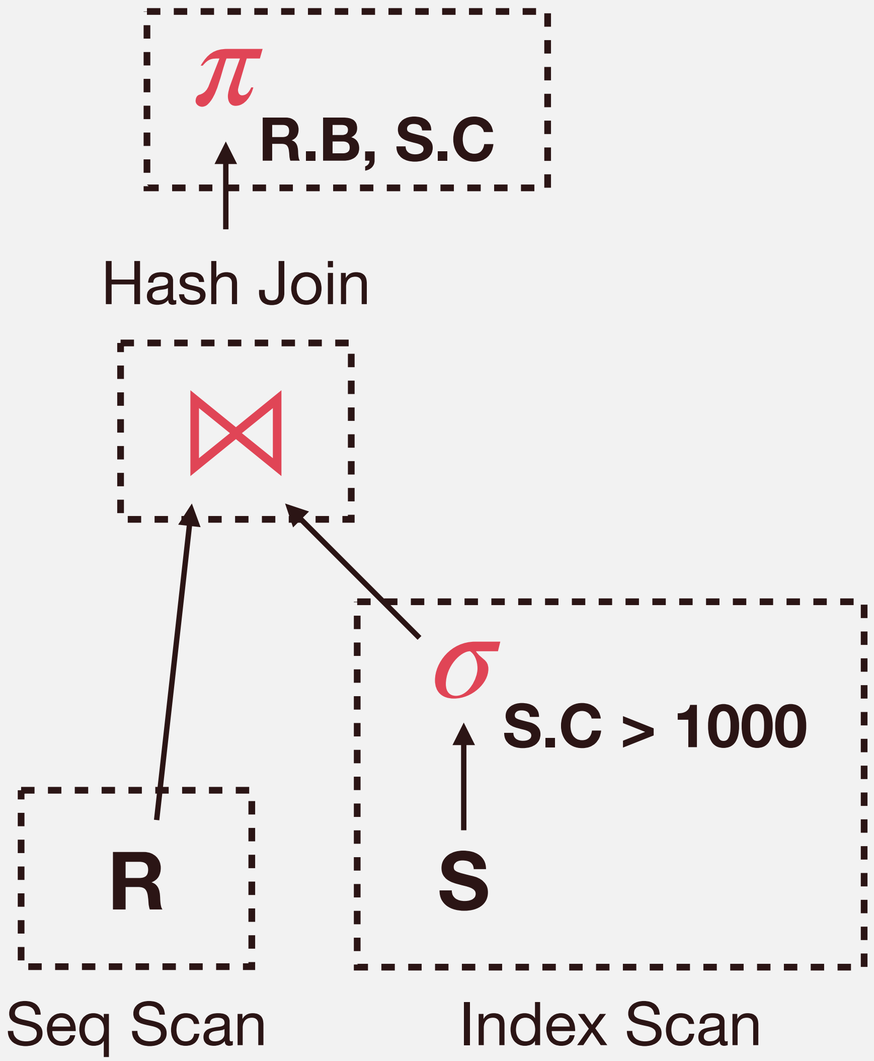
Iterator vs. Fully-Materialized

Iterator / Volcano	Fully-Materialized
Tuple at a time	Operator at a time
Small intermediate results	Need extra memory for intermediate results
A lot of virtual function calls	Fewer function calls
Can benefit from pipelining	Can benefit from batch processing (e.g, SIMD)
Adopted by almost every OLTP DBMS	A few DBMSs (e.g., monetDB, VoltDB)

Iterator vs. Fully-Materialized

Iterator / Volcano	Fully-Materialized
Tuple at a time	Operator at a time
Small intermediate results	Need extra memory for intermediate results
A lot of virtual function calls	Fewer function calls
Can benefit from pipelining	Can benefit from batch processing (e.g, SIMD)
Adopted by almost every OLTP DBMS	A few DBMSs (e.g., monetDB, VoltDB)

Vectorization Model



- ➜ Every operator implements **NextBatch( )**
  - Emits a batch of tuples
- ➜ Can use SIMD instructions in operator's internal loop to accelerate processing
- ➜ Much fewer function calls compared to the iterator model
- ➜ Ideal for OLAP
  - Adopted by most interpreted OLAP engines today

Execution Engine Design Space

➜ **Engine Type**

- **Interpretation**
- Compilation (Code-Gen)

➜ **Execution Model**

- Iterator / Volcano
- Fully-Materialized
- **Vectorization**

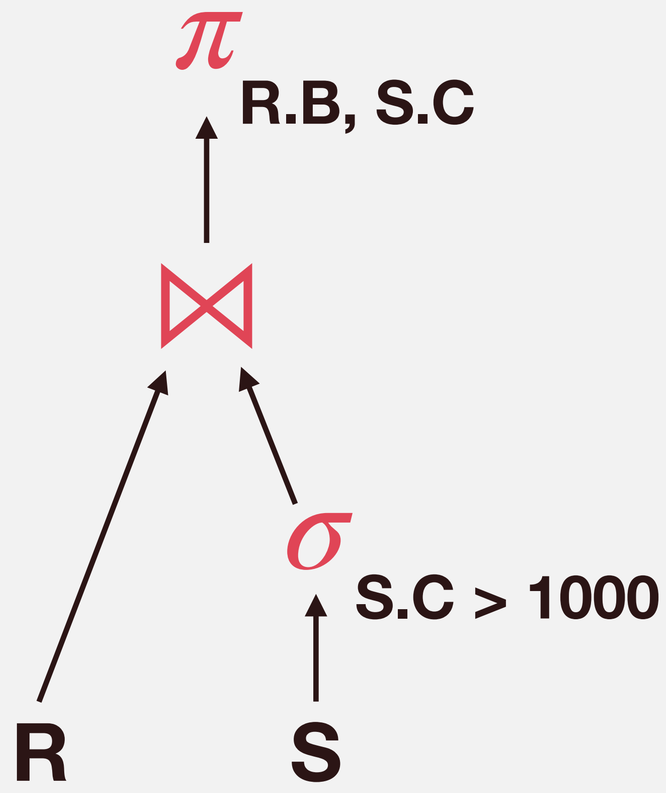
➜ **Pipeline Direction**

- Pull
- **Push**

Pipeline Direction

➔ **Pull**

- Parent operator “pulls” data up from its children
- Via function calls such as Next( )
- Most common way, easier to understand and implement



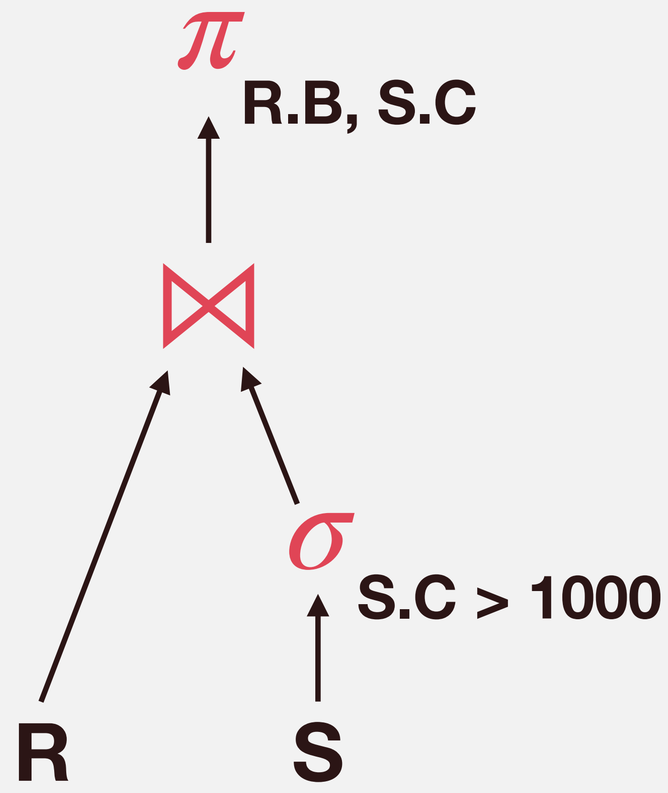
Pipeline Direction

➔ **Pull**

- Parent operator “pulls” data up from its children
- Via function calls such as Next( )
- Most common way, easier to understand and implement

➔ **Push**

- Child operator “pushes” data to its parent
- Similar to producer-consumer model
- Easier to “fuse” operations so that data stays in CPU register as long as possible



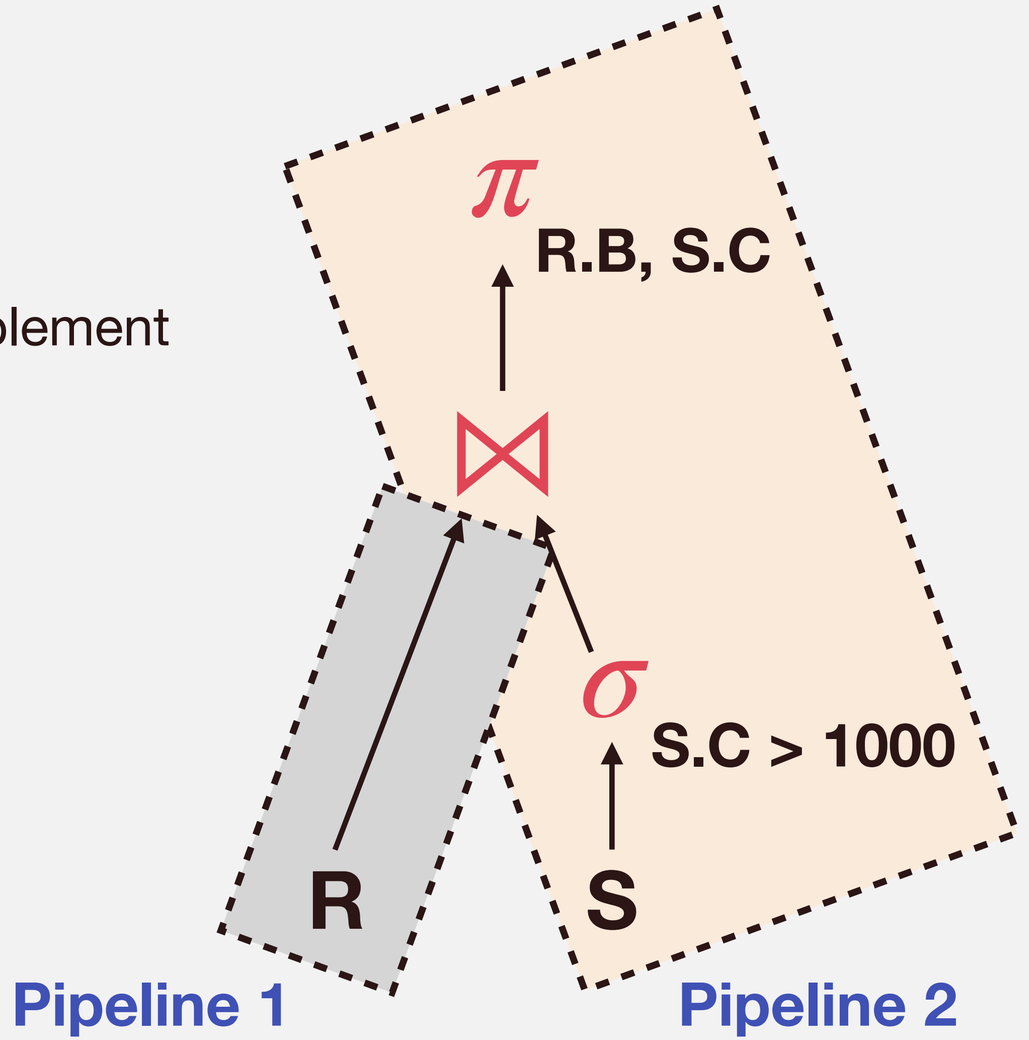
Pipeline Direction

➔ **Pull**

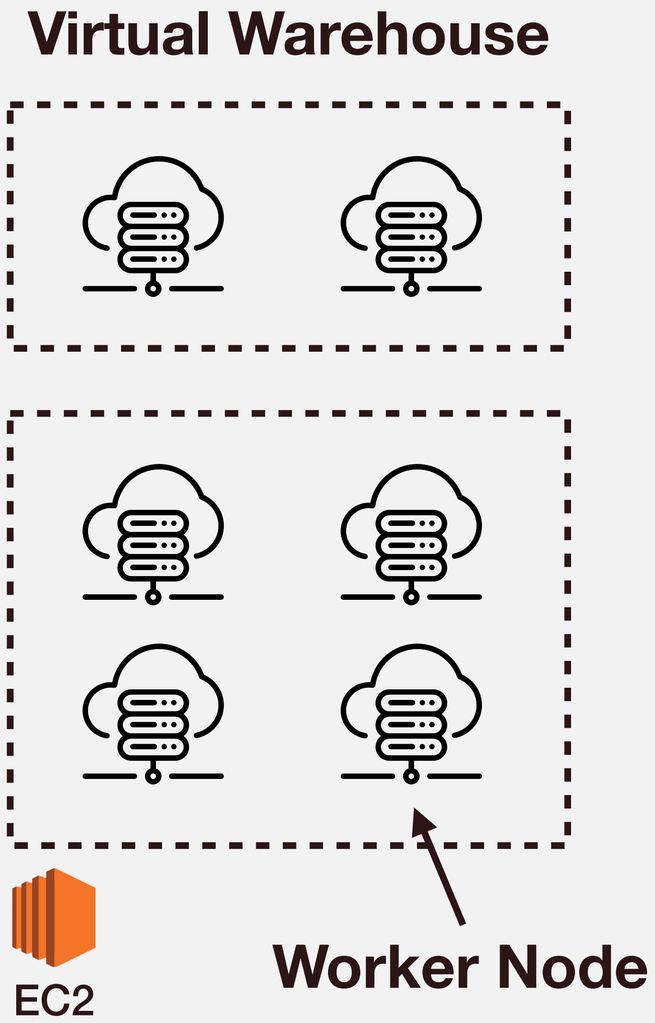
- Parent operator “pulls” data up from its children
- Via function calls such as Next( )
- Most common way, easier to understand and implement

➔ **Push**

- Child operator “pushes” data to its parent
- Similar to producer-consumer model
- Easier to “fuse” operations so that data stays in CPU register as long as possible
- **Centralized control on task scheduling**

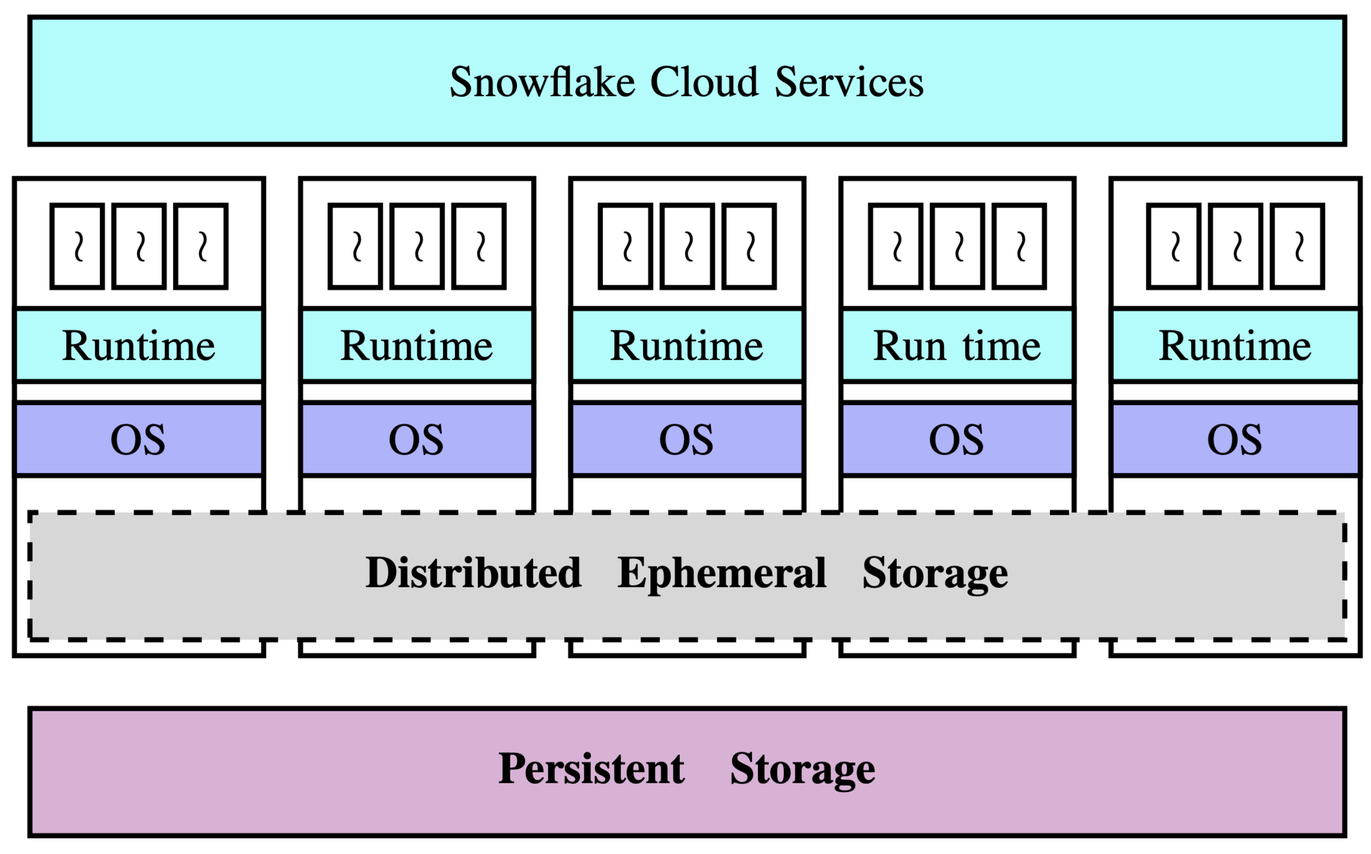


Virtual Warehouses: the Muscle

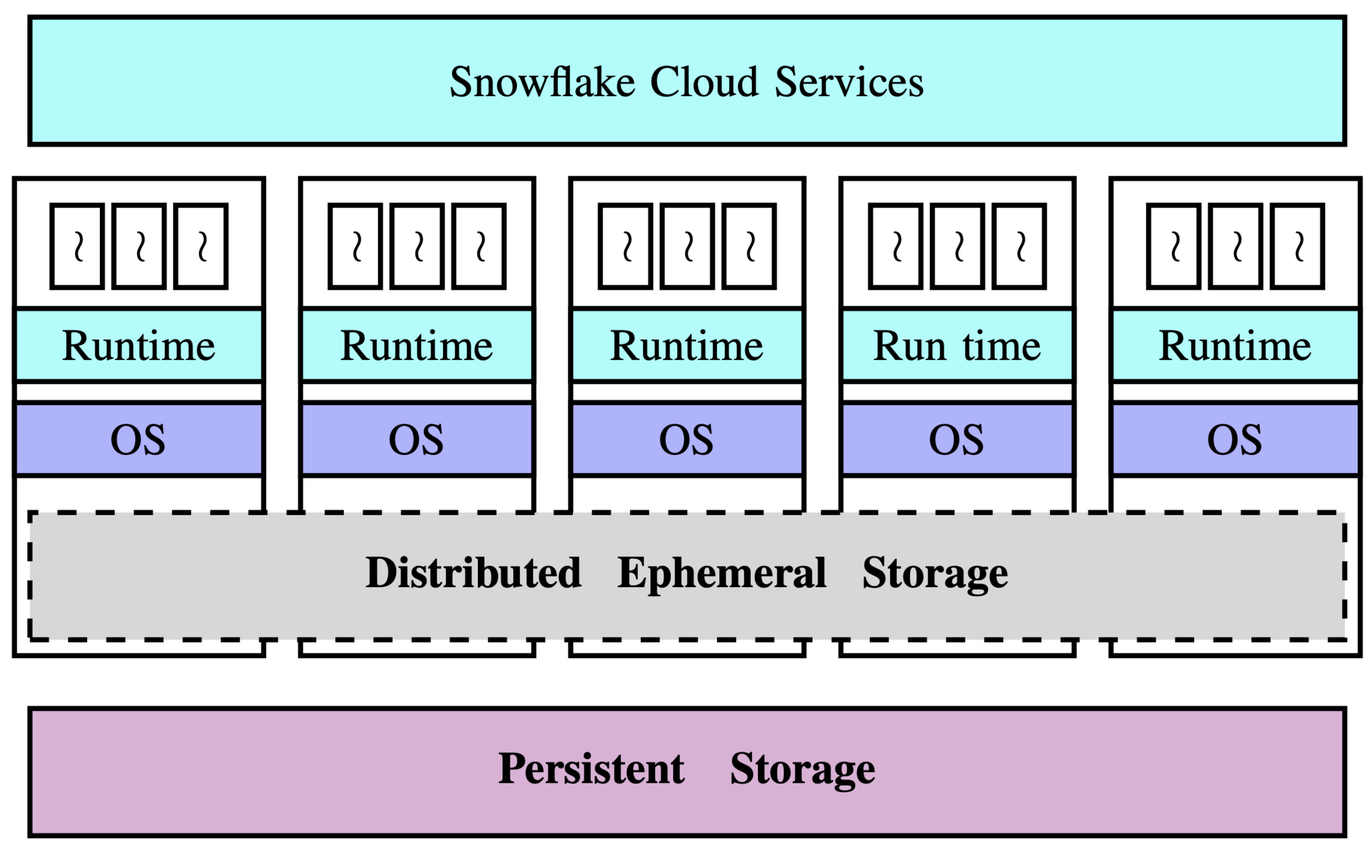


- ➜ Create, destroy, resize on demand
- ➜ Performance Isolation
  - Shared data, private compute
  - Typical usage pattern
    - Continuously-running VWs for repeating jobs
    - On-demand VWs for ad-hoc tasks
- ➜ Ephemeral worker processes
- ➜ **Columnar, Vectorized, Push-Based**

Ephemeral Storage System



Ephemeral Storage System



➜ Intermediate Results Spilling

Memory

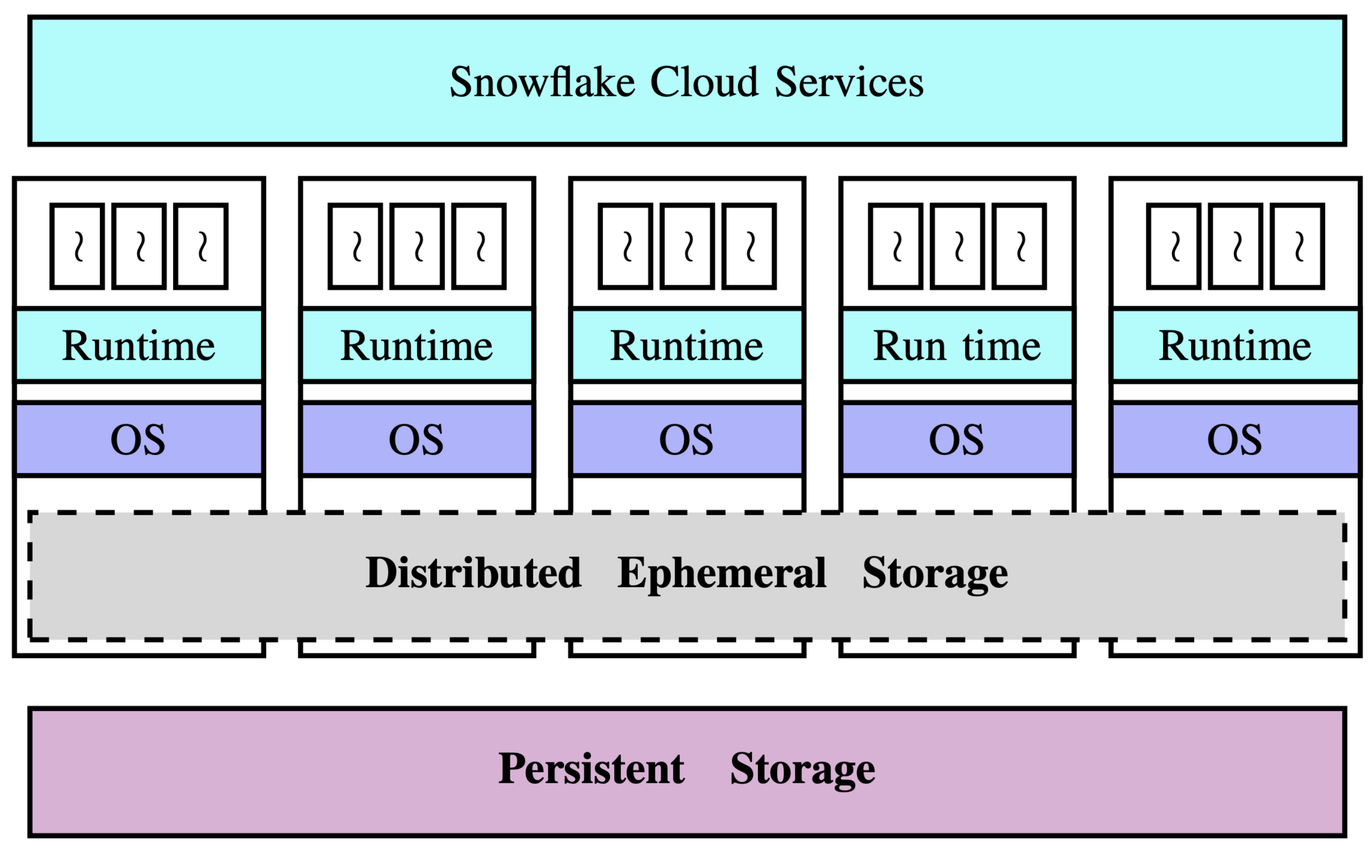
↓

Local SSD

↓

S3

Ephemeral Storage System



➜ Intermediate Results Spilling

Memory

↓

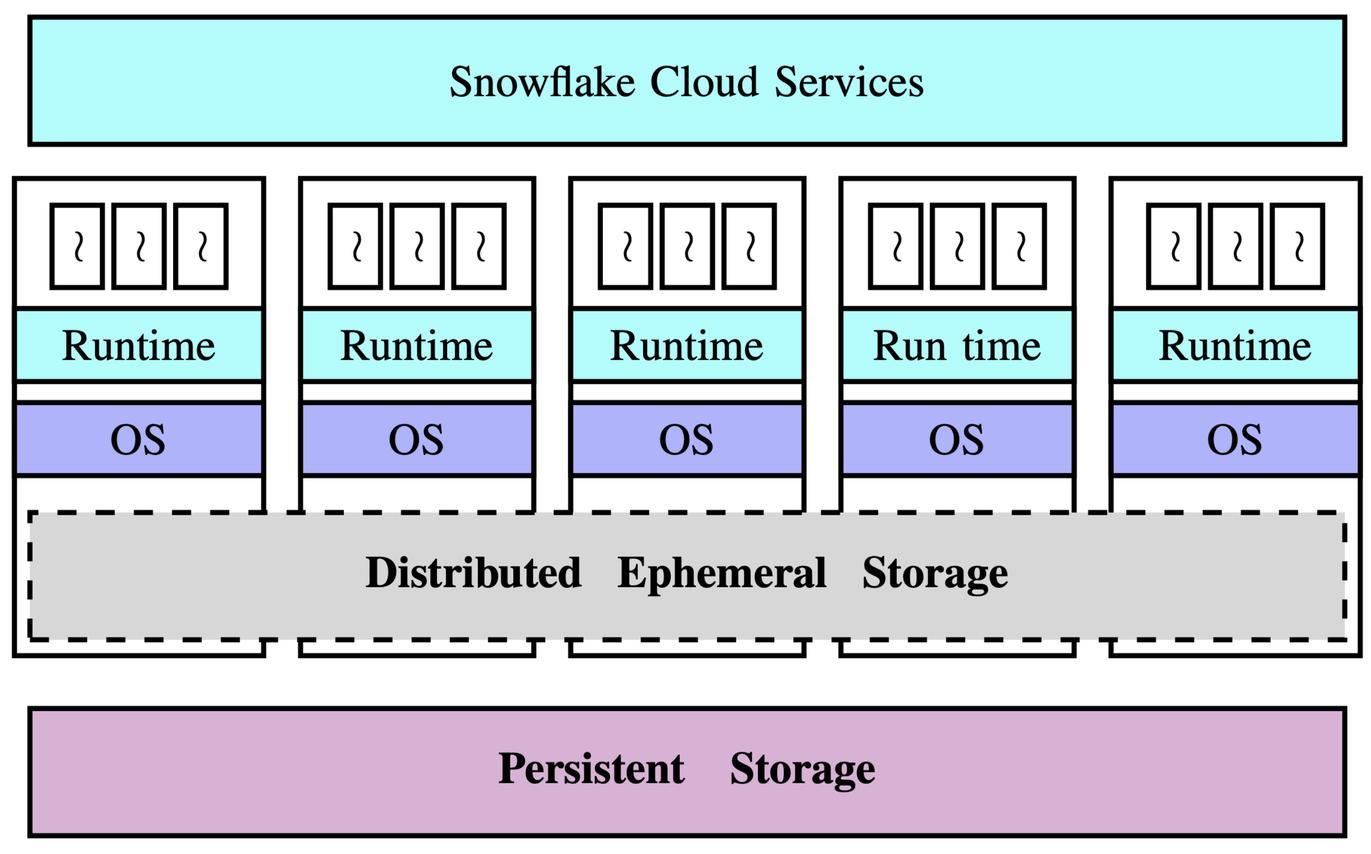
Local SSD

↓

S3

Hard to avoid unless Ephemeral Storage is a separate fully-elastic service

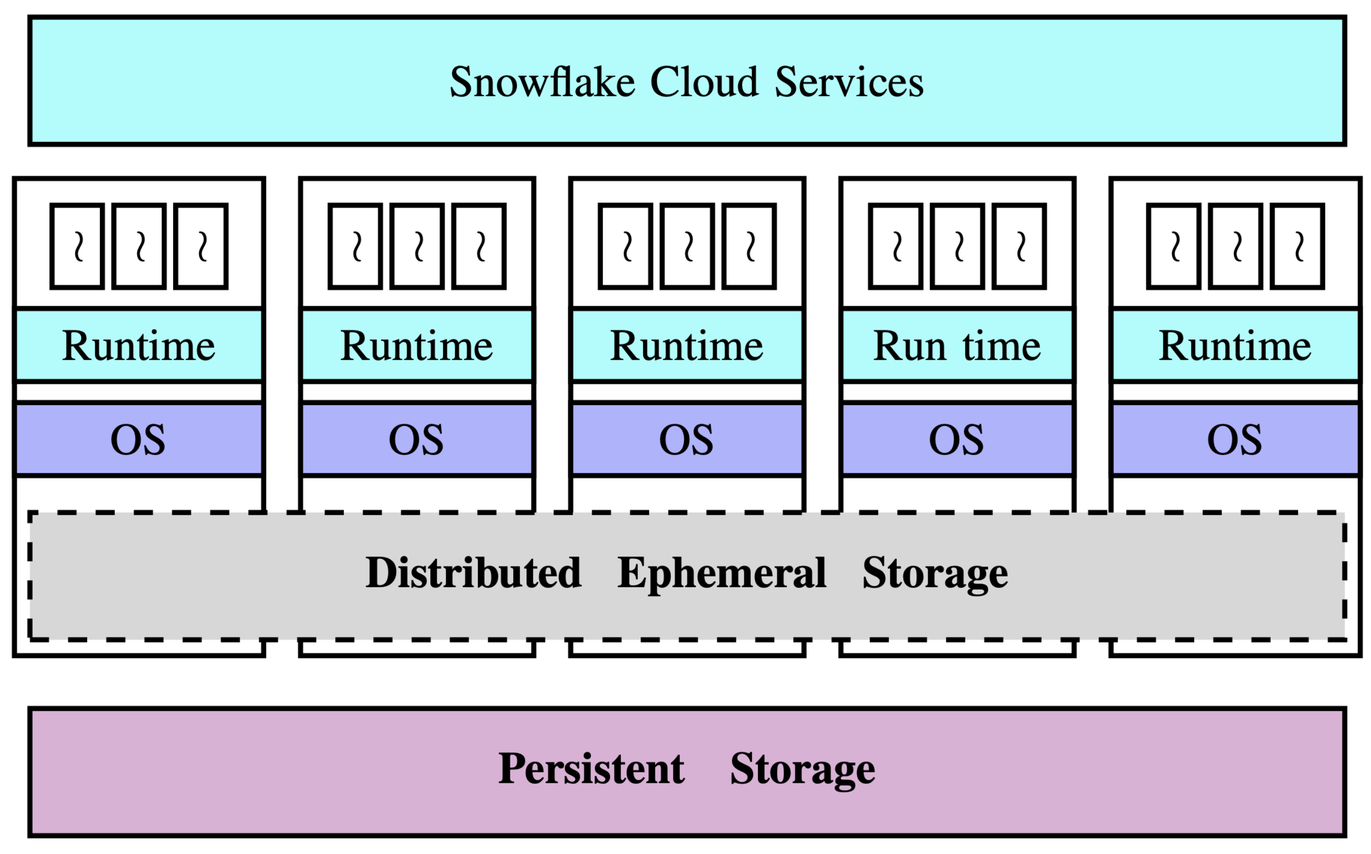
Ephemeral Storage System



➜ **Lazy Consistency Hashing**

- Use CH to cache persistent files
- Optimizer assign scan set using the same CH

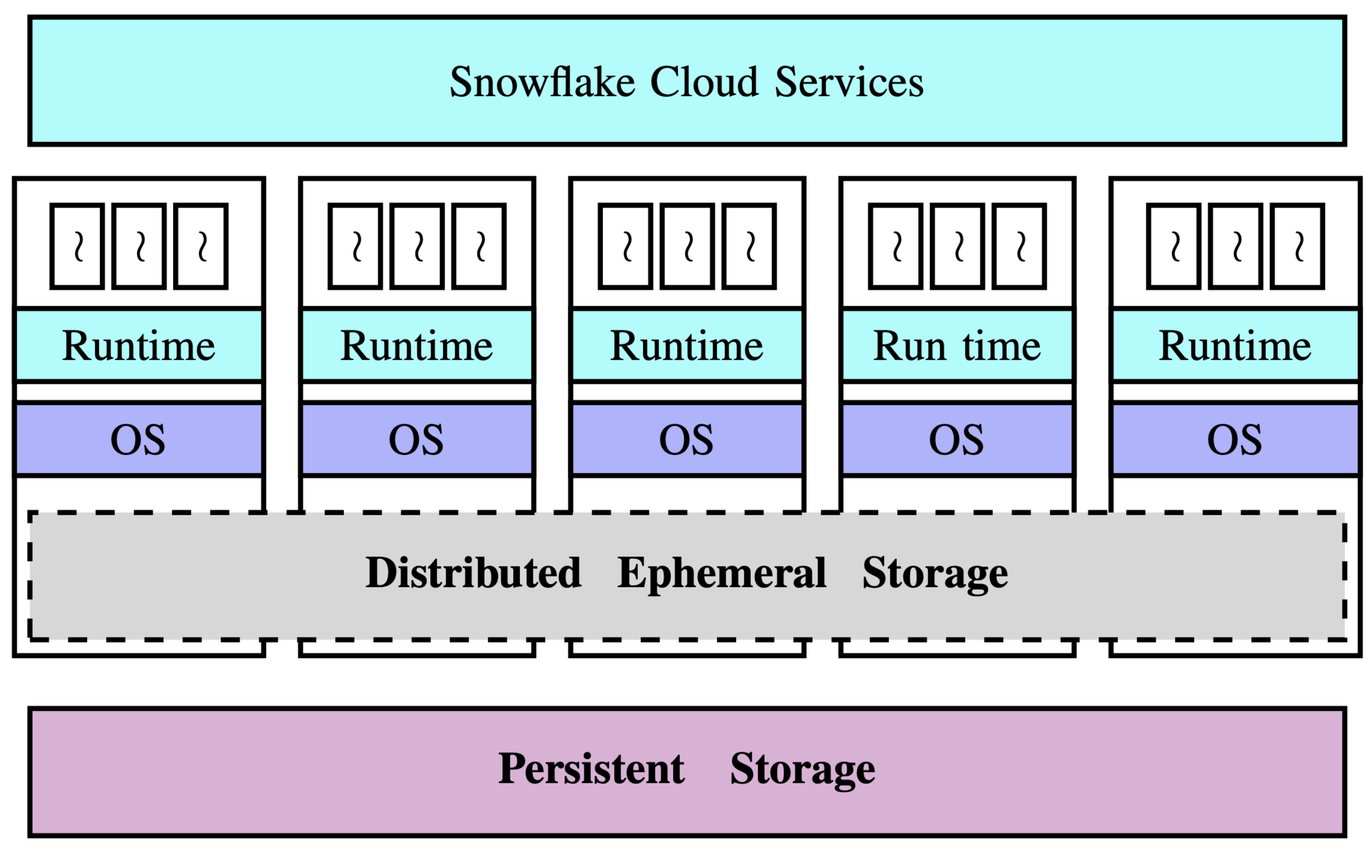
Ephemeral Storage System



➜ **Lazy Consistency Hashing**

- Use CH to cache persistent files
- Optimizer assign scan set using the same CH
- Lazy: no reshuffling

Ephemeral Storage System



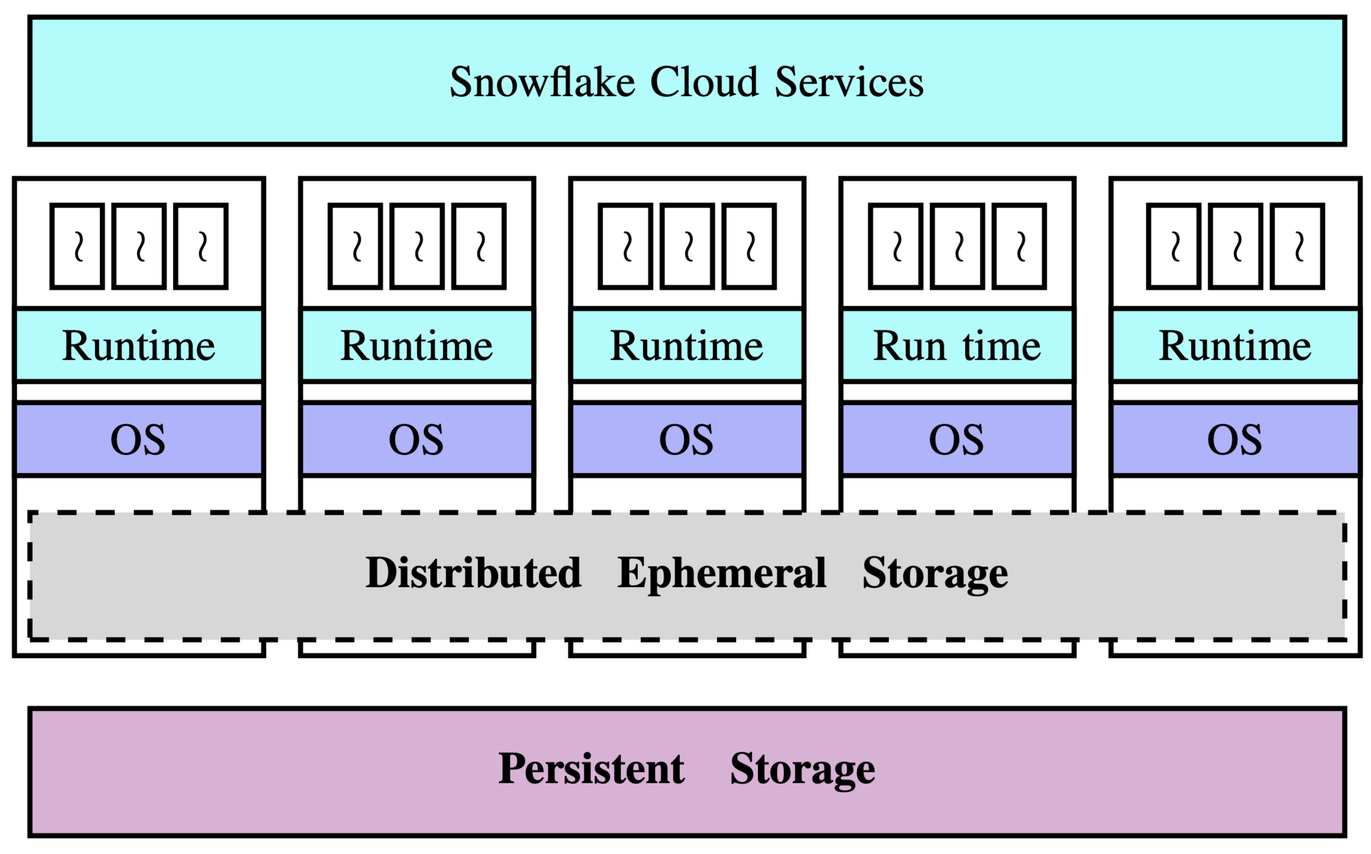
➜ **Lazy Consistency Hashing**

- Use CH to cache persistent files
- Optimizer assign scan set using the same CH
- Lazy: no reshuffling

➜ **“Opportunistically” Caching**

- Intermediate results always have higher priority than cached files

Ephemeral Storage System



➜ **Lazy Consistency Hashing**

- Use CH to cache persistent files
- Optimizer assign scan set using the same CH
- Lazy: no reshuffling

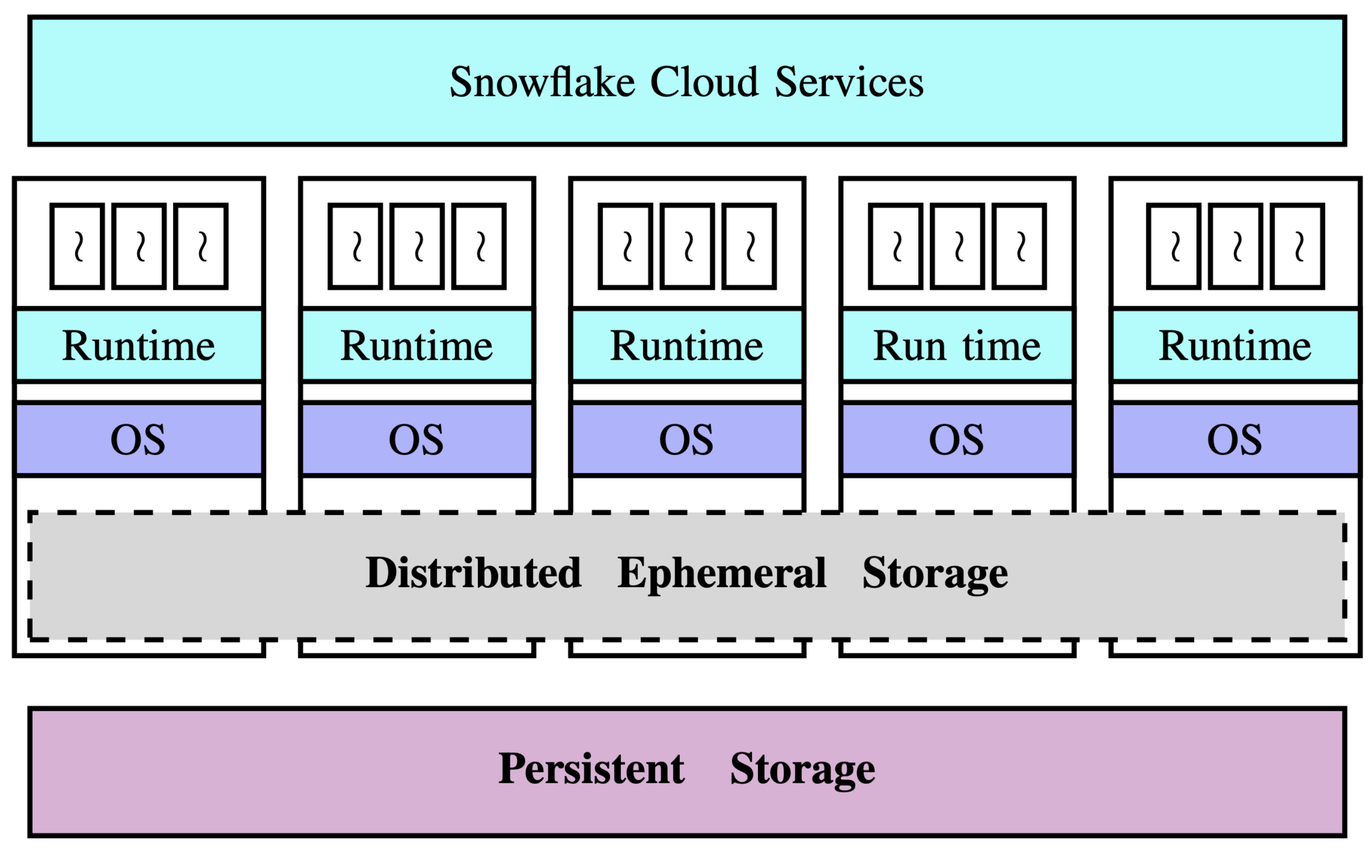
➜ **“Opportunistically” Caching**

- Intermediate results always have higher priority than cached files

➜ **File Stealing**

- “Stealer” download files from S3, not the “straggler”

Ephemeral Storage System



What if a worker fails during the execution?

The query has to restart

➜ **Lazy Consistency Hashing**

- Use CH to cache persistent files
- Optimizer assign scan set using the same CH
- Lazy: no reshuffling

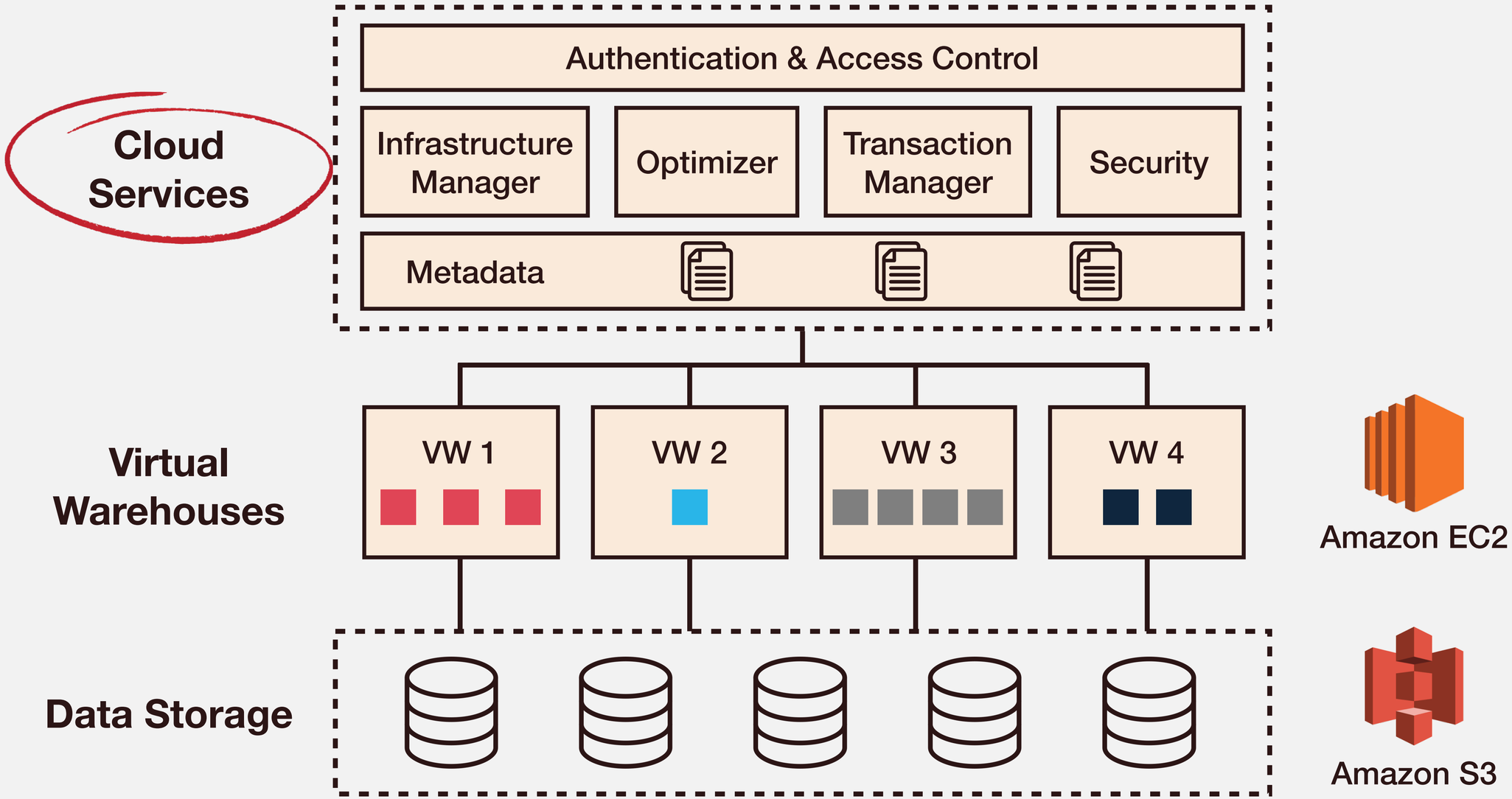
➜ **“Opportunistically” Caching**

- Intermediate results always have higher priority than cached files

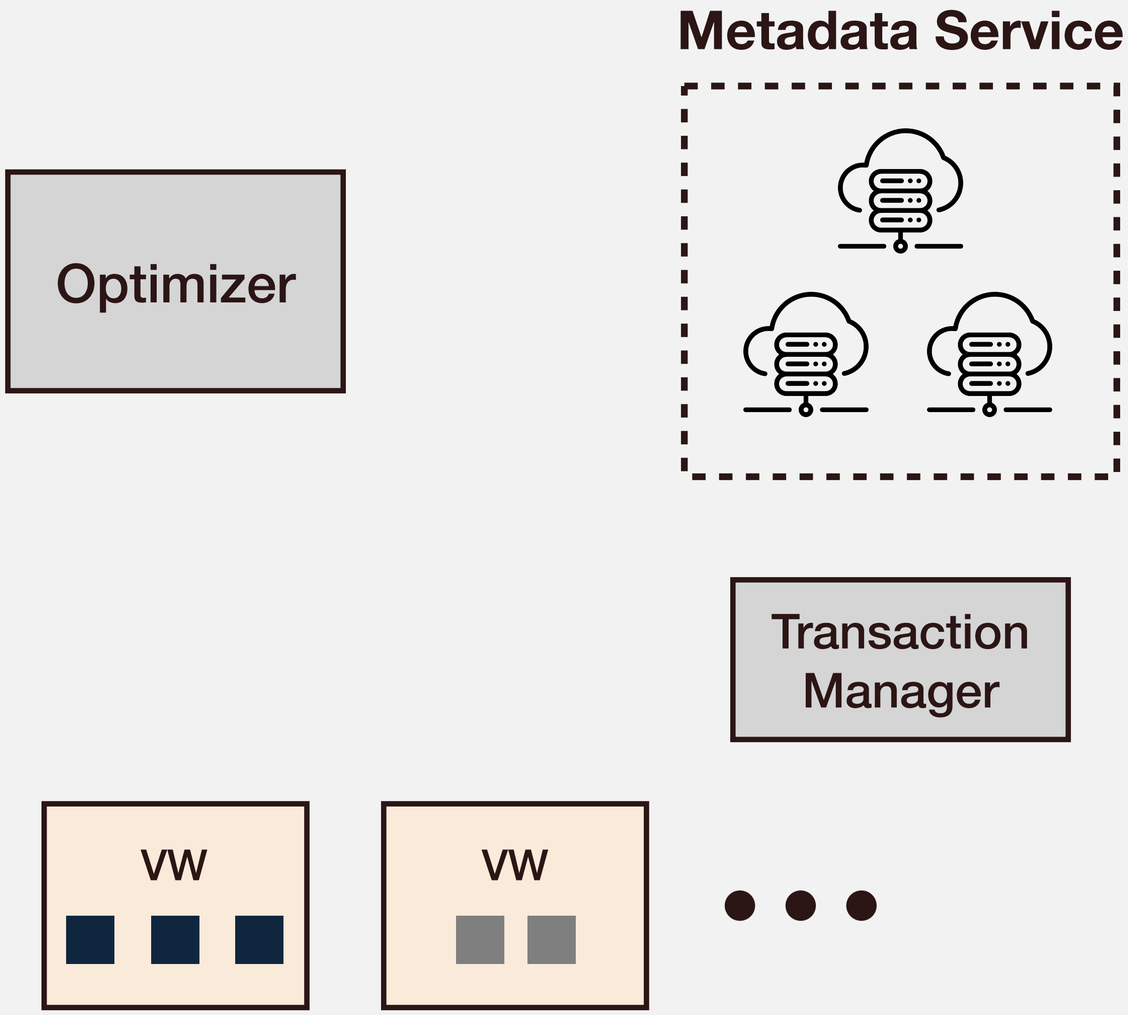
➜ **File Stealing**

- “Stealer” download files from S3, not the “straggler”

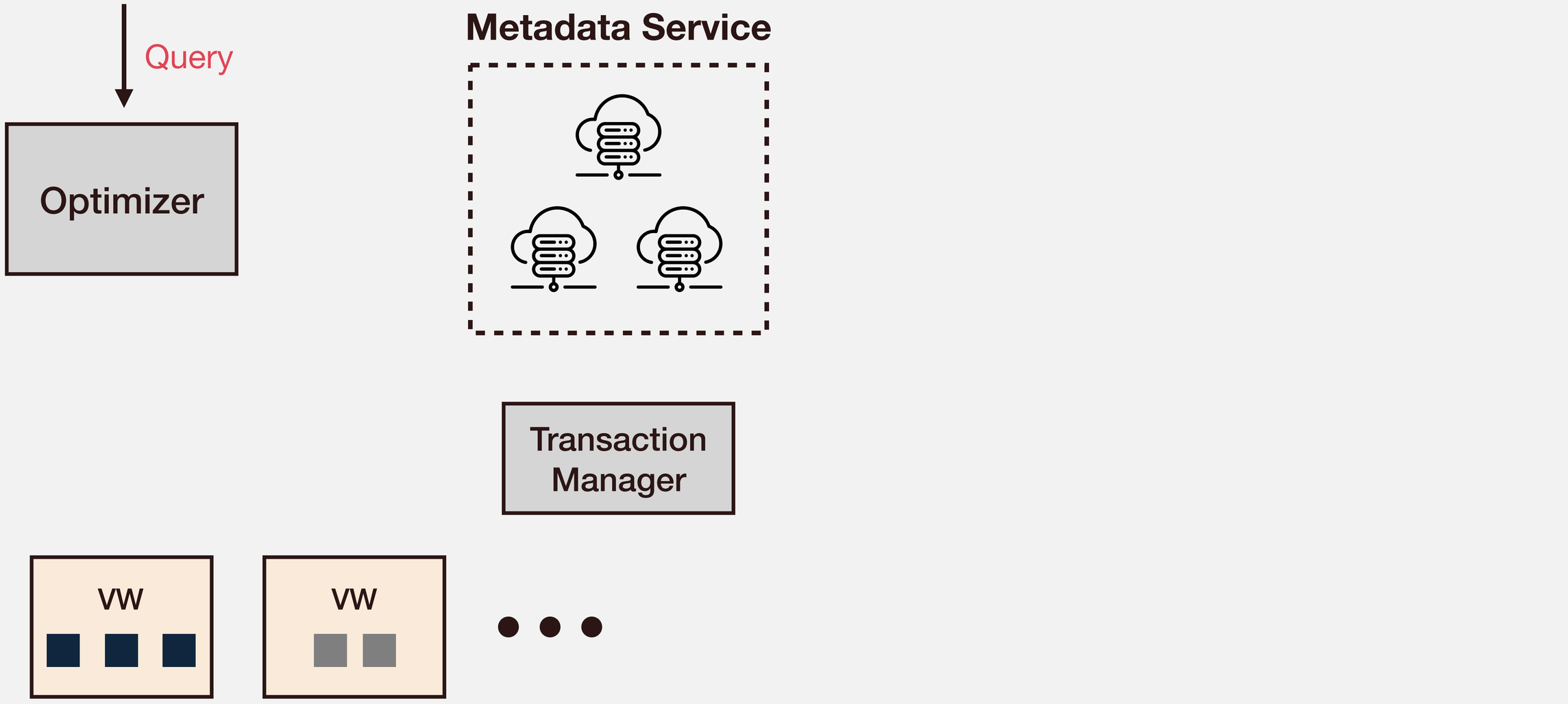
Snowflake Architecture



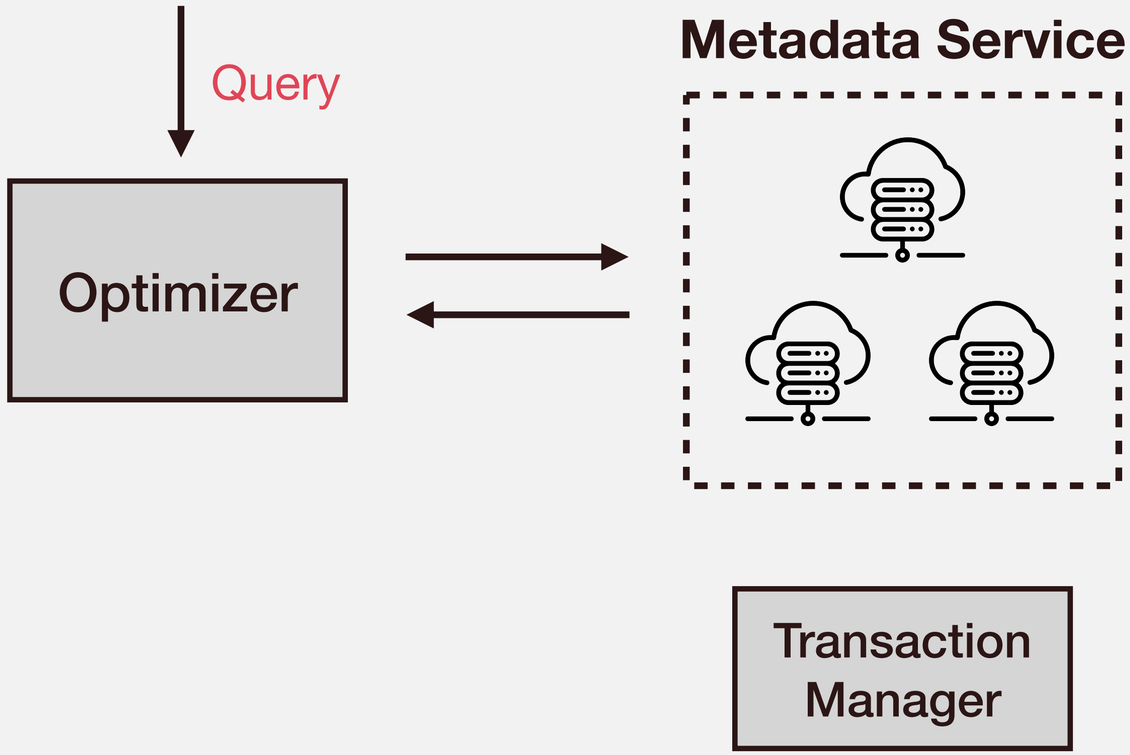
Cloud Services: the Brain



Cloud Services: the Brain



Cloud Services: the Brain



➜ **Optimizer**

- Cascade-style
- Scan set **pruning**

Pruning and Reclustering

How to avoid full table scan?

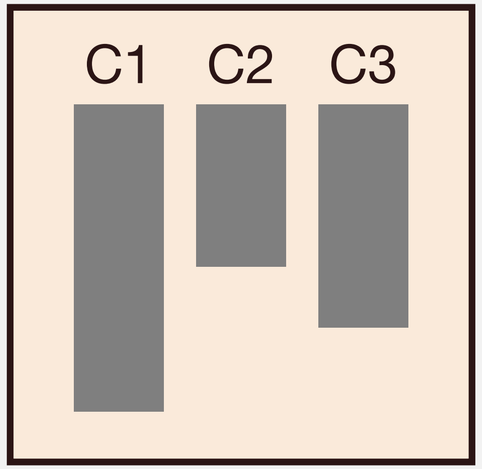
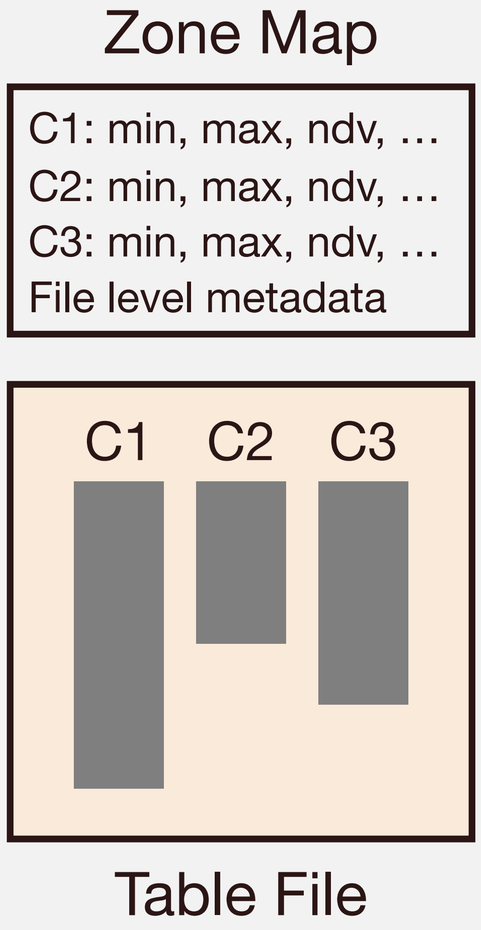


Table File

Pruning and Reclustering

How to avoid full table scan?

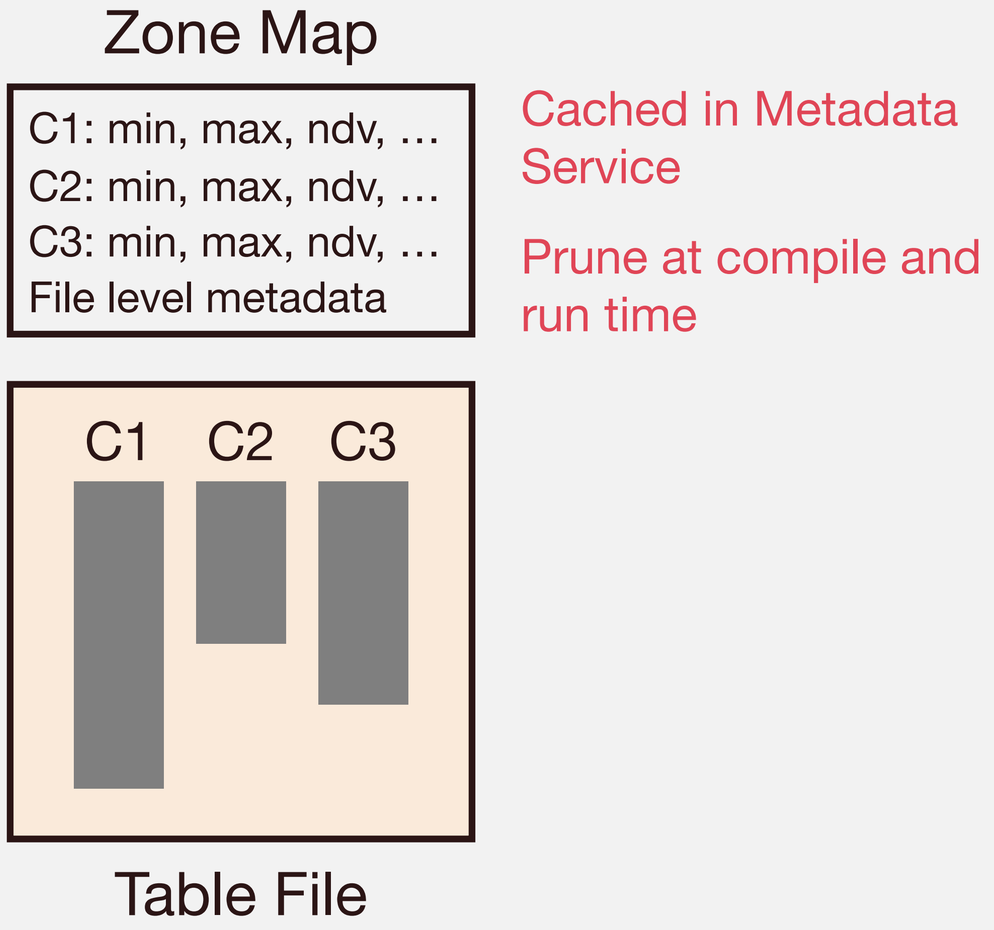


Cached in Metadata Service

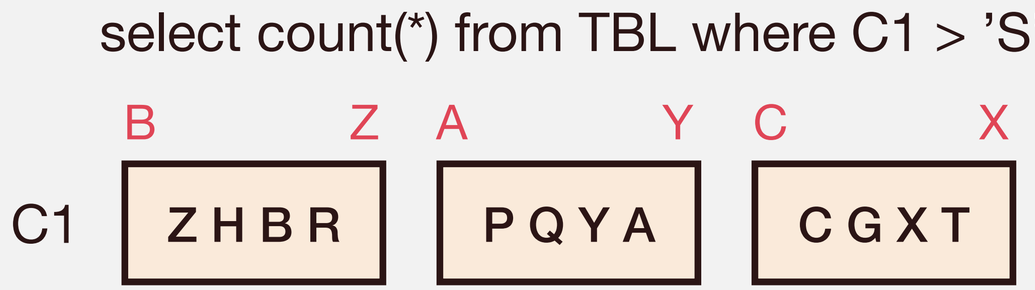
Prune at compile and run time

Pruning and Reclustering

How to avoid full table scan?



But it only works with data locality



Pruning and Reclustering

How to avoid full table scan?

Zone Map

C1: min, max, ndv, …
C2: min, max, ndv, …
C3: min, max, ndv, …
File level metadata

Cached in Metadata Service

Prune at compile and run time

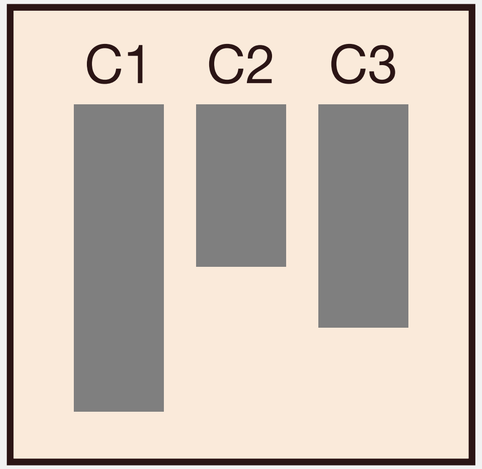


Table File

But it only works with data locality

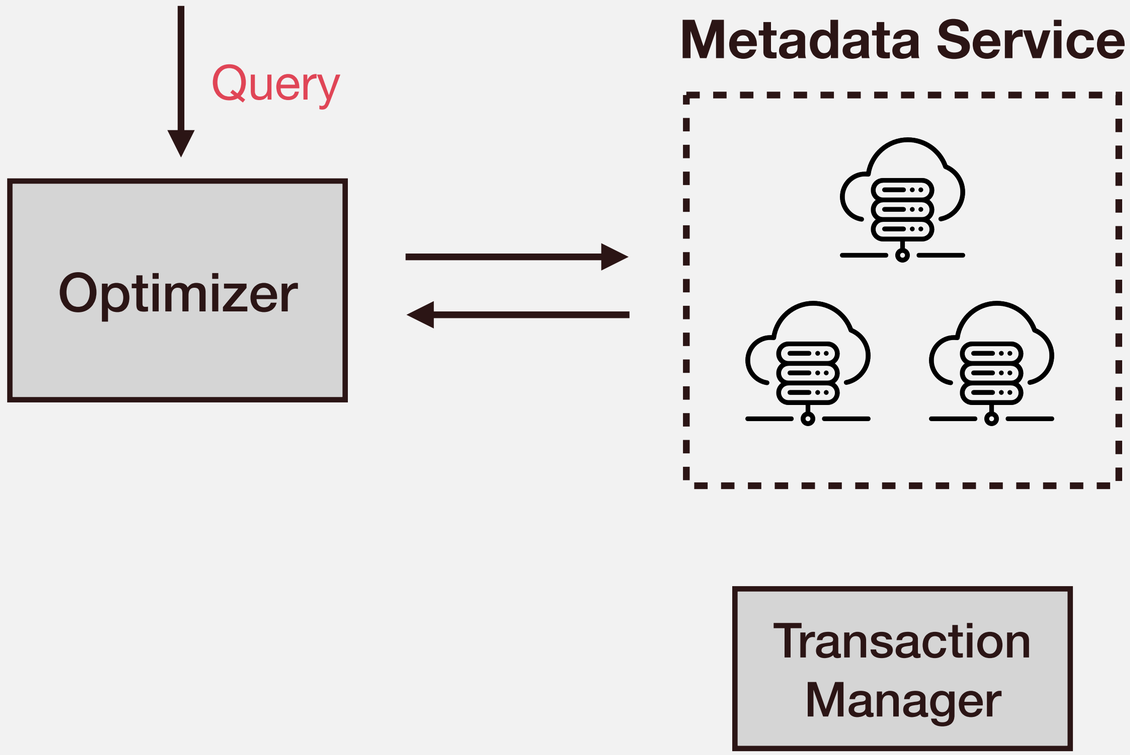
select count(*) from TBL where C1 > 'S'

Reclustering

Keep data "mostly" sorted

Automatic, incremental in the background

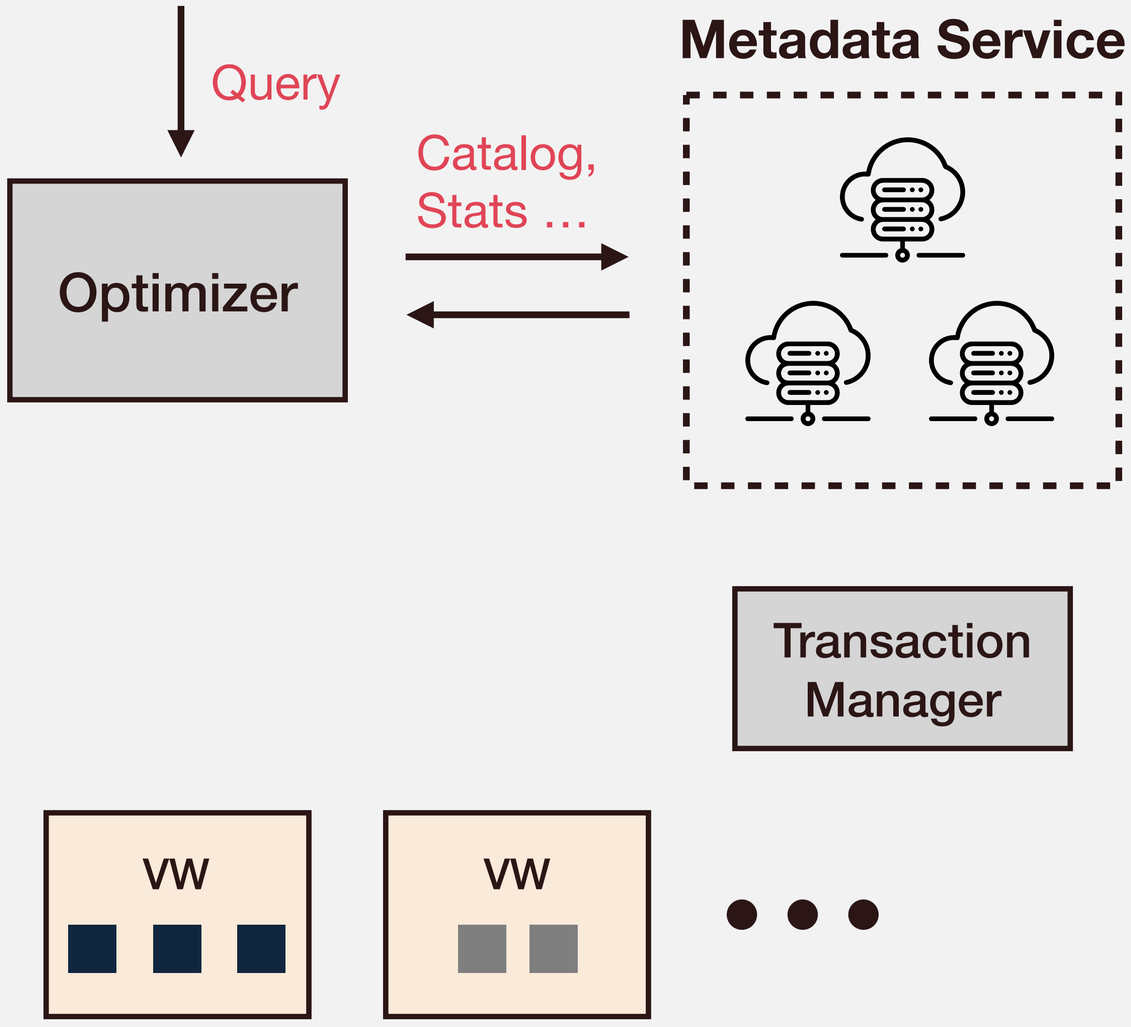
Cloud Services: the Brain



➜ **Optimizer**

- Cascade-style
- Scan set **pruning**

Cloud Services: the Brain



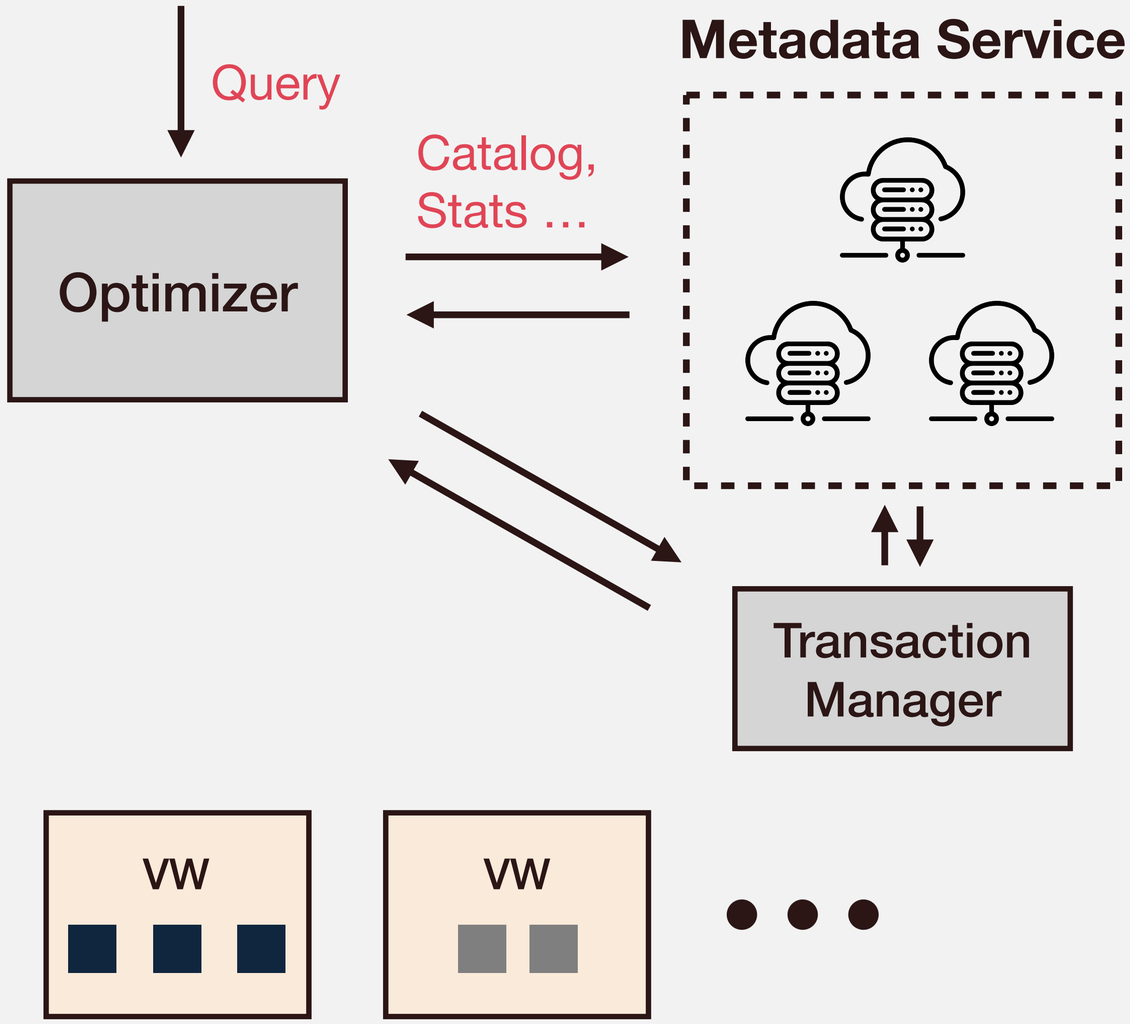
➜ **Optimizer**

- Cascade-style
- Scan set **pruning**

➜ **Metadata Service**

- Stand-alone **FoundationDB** cluster for low latency accesses
- Info needed for query compilation
  - Catalog, Stats
  - Lock status, version info
  - Zone maps

Cloud Services: the Brain



➔ **Optimizer**

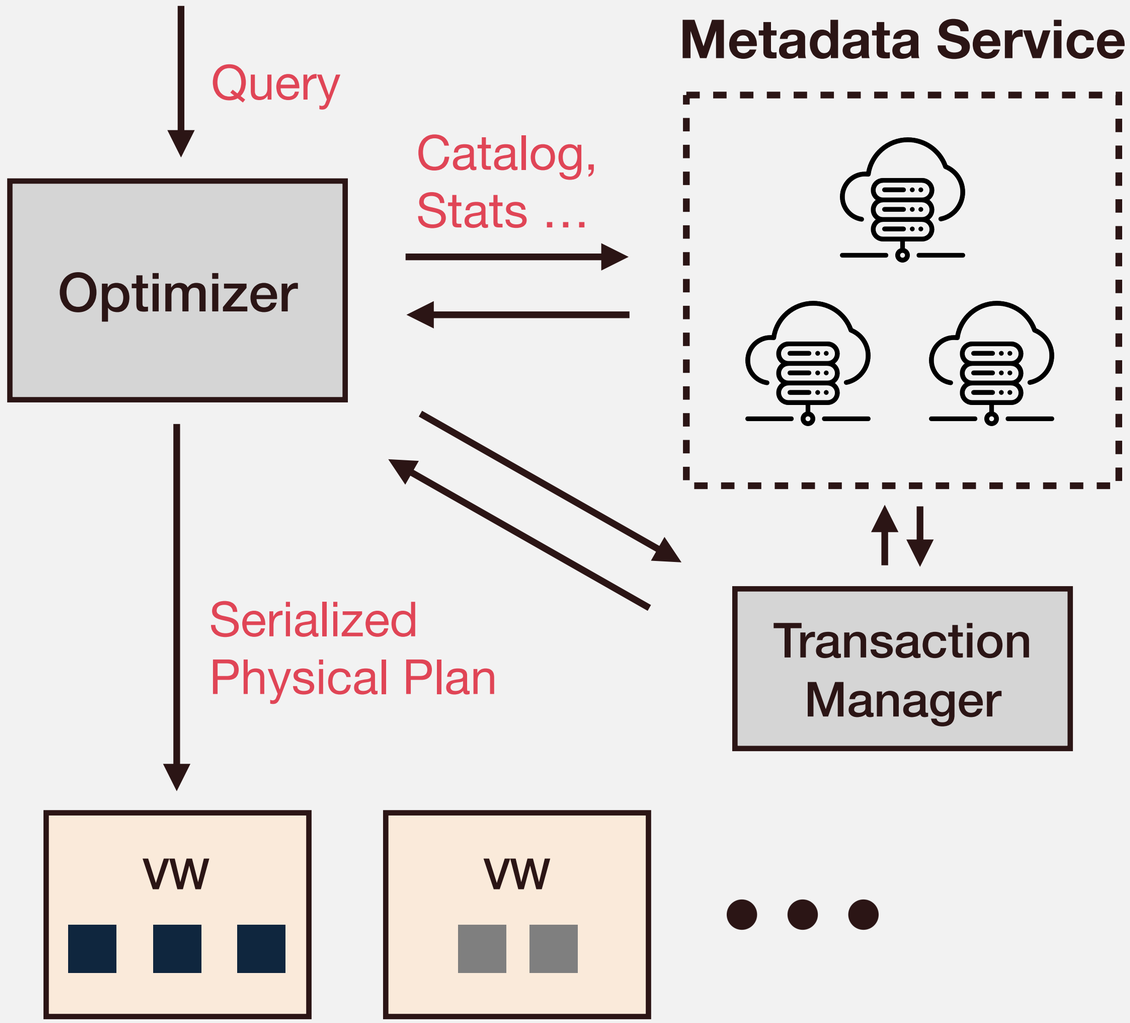
- Cascade-style
- Scan set **pruning**

➔ **Metadata Service**

- Stand-alone **FoundationDB** cluster for low latency accesses
- Info needed for query compilation
  - Catalog, Stats
  - Lock status, version info
  - Zone maps

➔ **MVCC (Snapshot Isolation)**

Cloud Services: the Brain



➔ **Optimizer**

- Cascade-style
- Scan set **pruning**

➔ **Metadata Service**

- Stand-alone **FoundationDB** cluster for low latency accesses
- Info needed for query compilation
  - Catalog, Stats
  - Lock status, version info
  - Zone maps

➔ **MVCC (Snapshot Isolation)**

Snowflake Architecture Summary

- ➜ Disaggregated compute and storage
- ➜ Immutable hybrid columnar files in object storage
- ➜ Virtual warehouses provide elasticity and performance isolation
- ➜ Vectorized push-based execution engine
- ➜ Ephemeral storage system for caching intermediate results and persistent files
- ➜ Multi-tenant, always-on cloud services
- ➜ Separate fast metadata store
- ➜ Cascades-style optimizer, zone maps for scan pruning