

CS4221

Cloud Databases I. Fundamentals

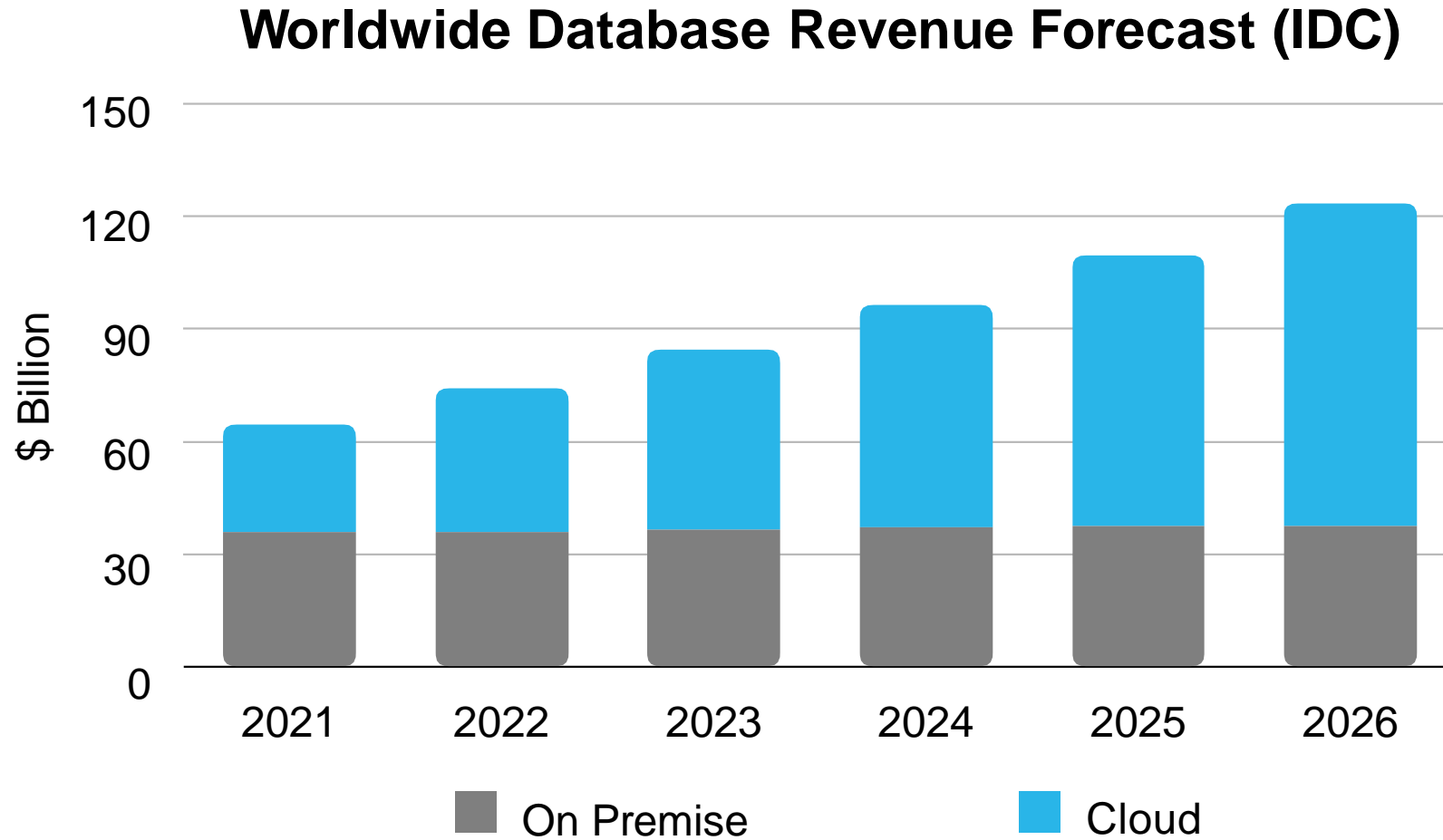
Yao LU
2024 Semester 2

National University of Singapore
School of Computing

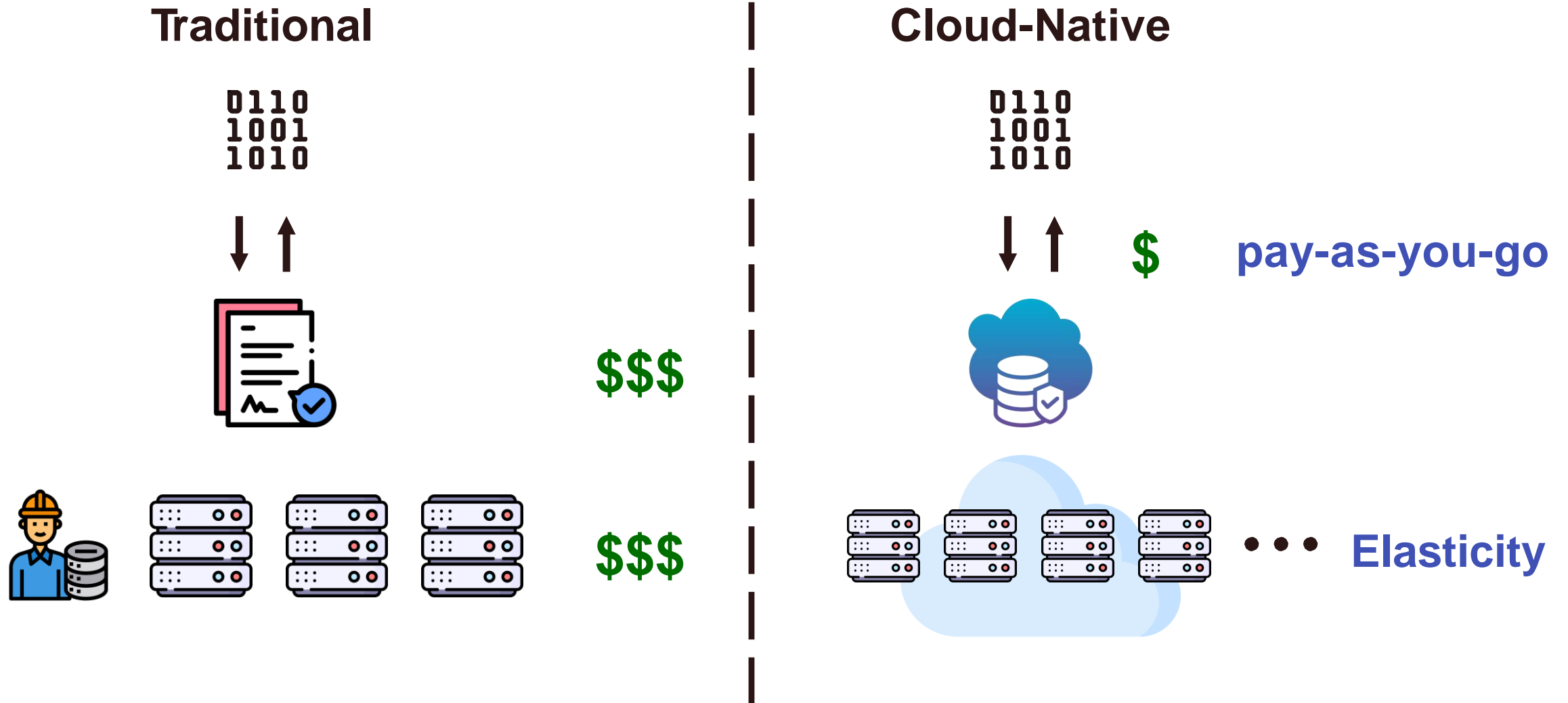
Cloud databases: outline

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

The booming cloud database market



Why are cloud databases different?



Traditional vs. cloud databases

	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 +

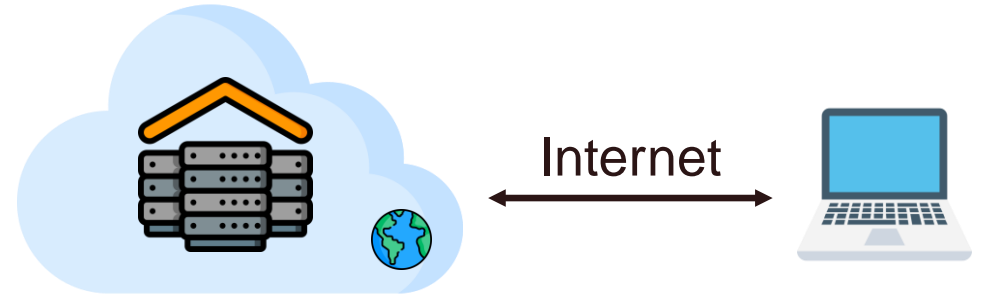
Cloud computing

- 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

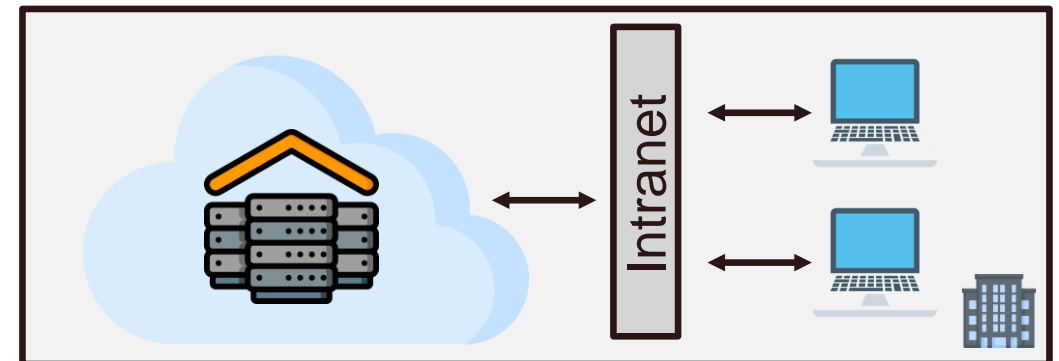
- **Public Cloud**

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



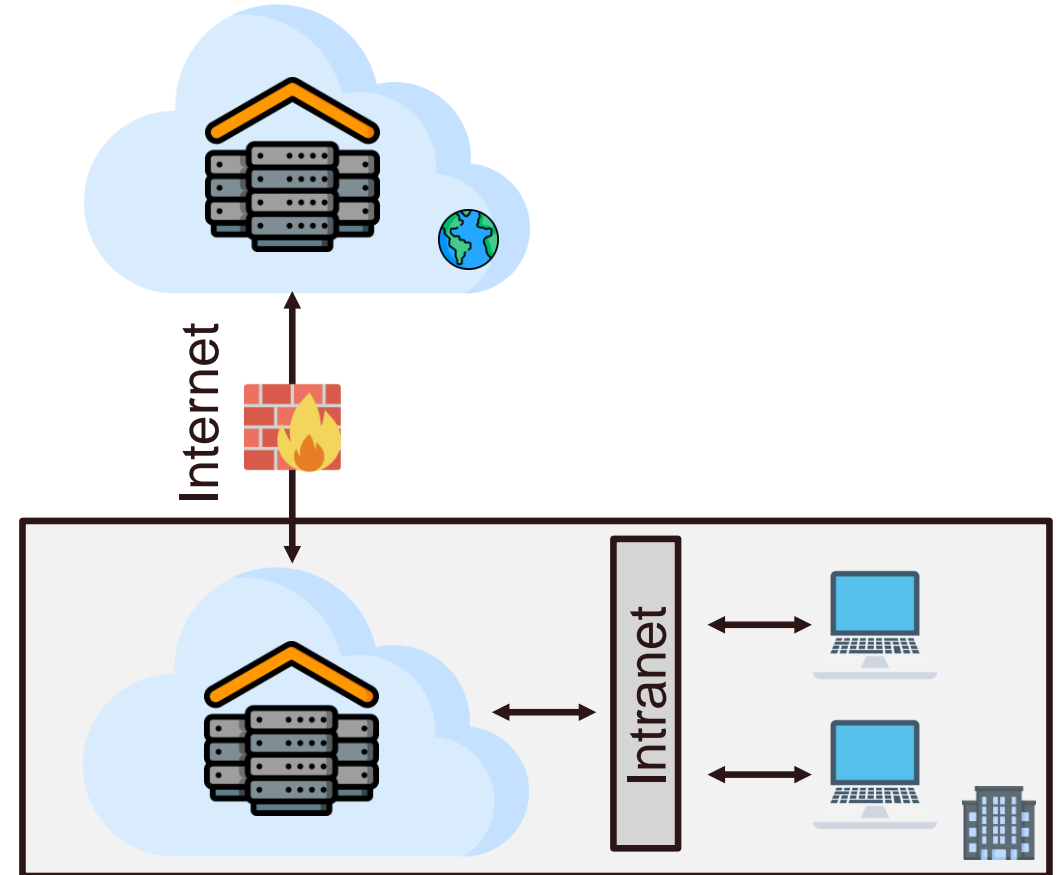
- **Private Cloud**

- 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)



Cloud building blocks

Applications

E.g., Email, Google Drive

Development Platform

E.g., Google App Engine, AWS Lambda

Resource Sharing

Virtualizing Resources

Infrastructure

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



Amazon
EC2



Amazon
S3



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
 - **Limitations**
 - Vendor lock-in
 - Hard to customize
 - Data security



AWS services

Storage:



S3



EBS

Compute:

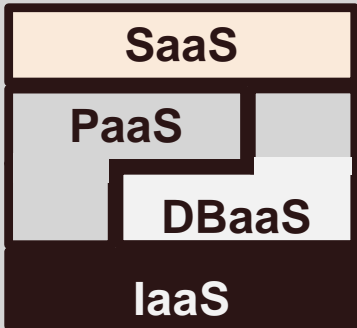


EC2



Lambda

Database:



RDS
Aurora

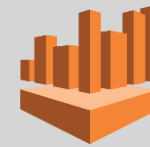


Amazon Redshift



amazon
DynamoDB

Big Data:



Athena



amazon
EMR



Amazon
SageMaker



AWS Simple Storage Service (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



AWS 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
- + Super cheap



S3

- **Ideal Use Case**

- Large data volume
- Data mostly immutable
- Latency-insensitive app

AWS 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



EC2

AWS Lambda

- **Serverless Computing**

“... 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

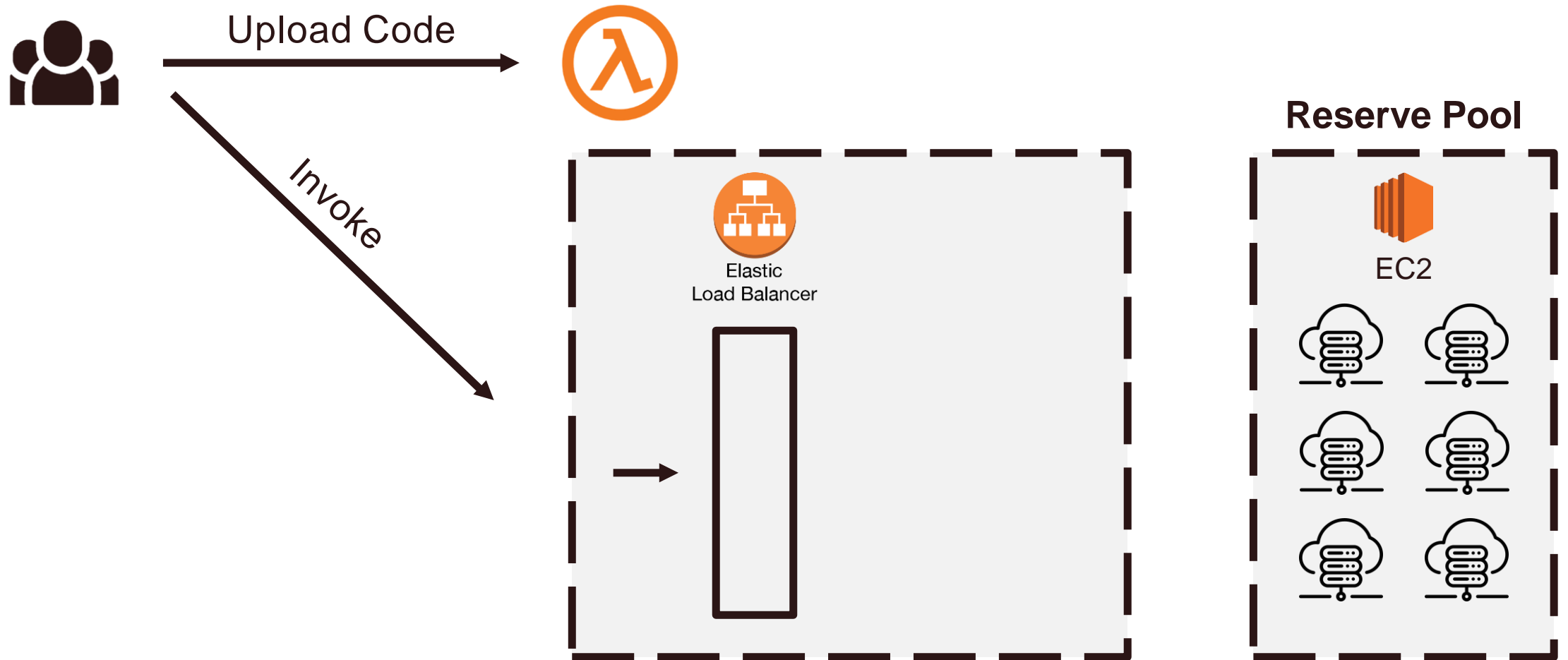


Lambda

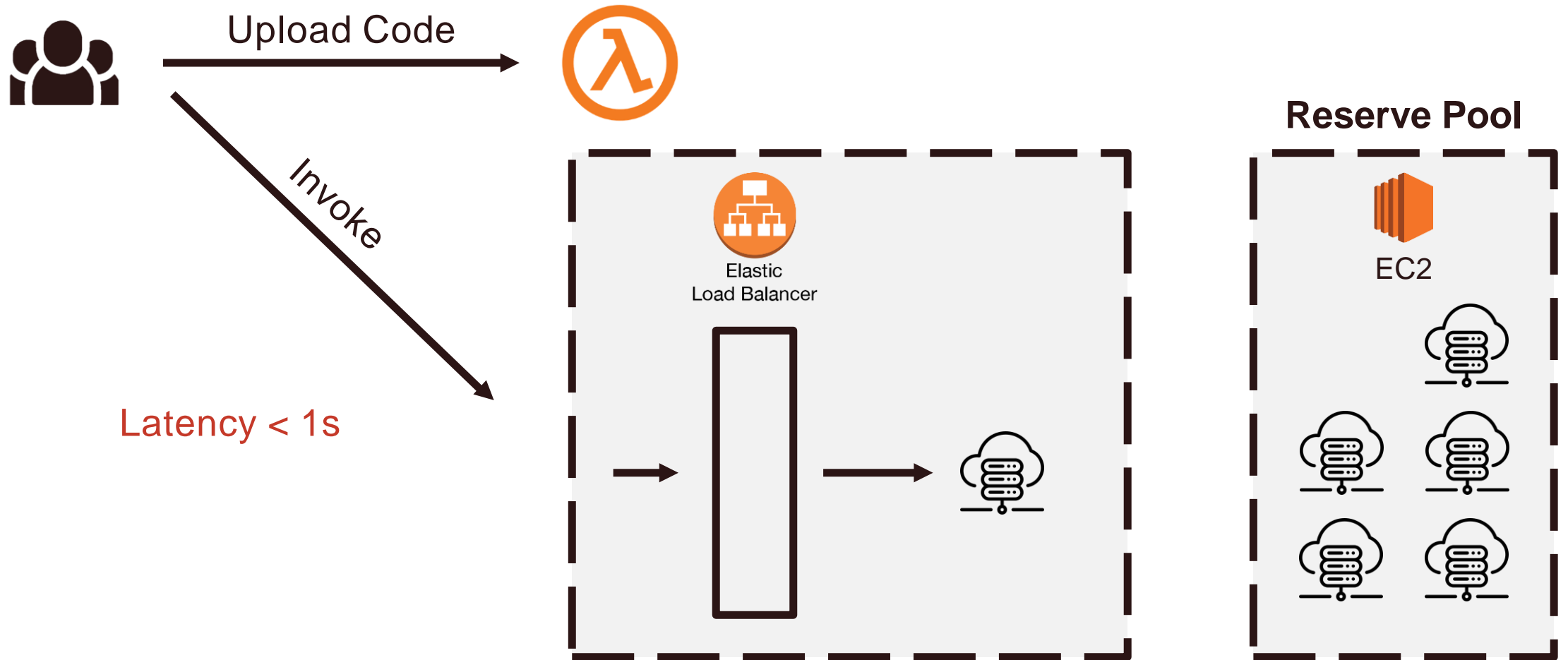
AWS Lambda



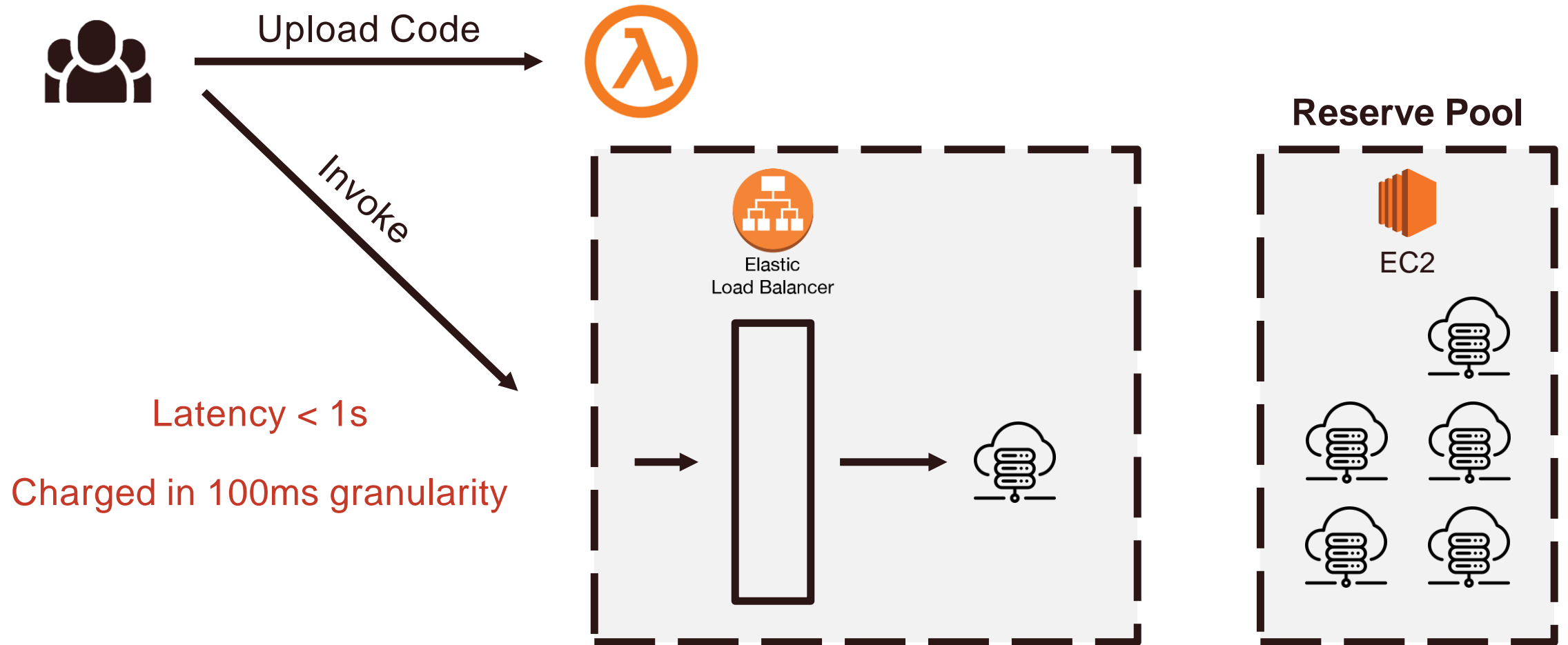
AWS Lambda



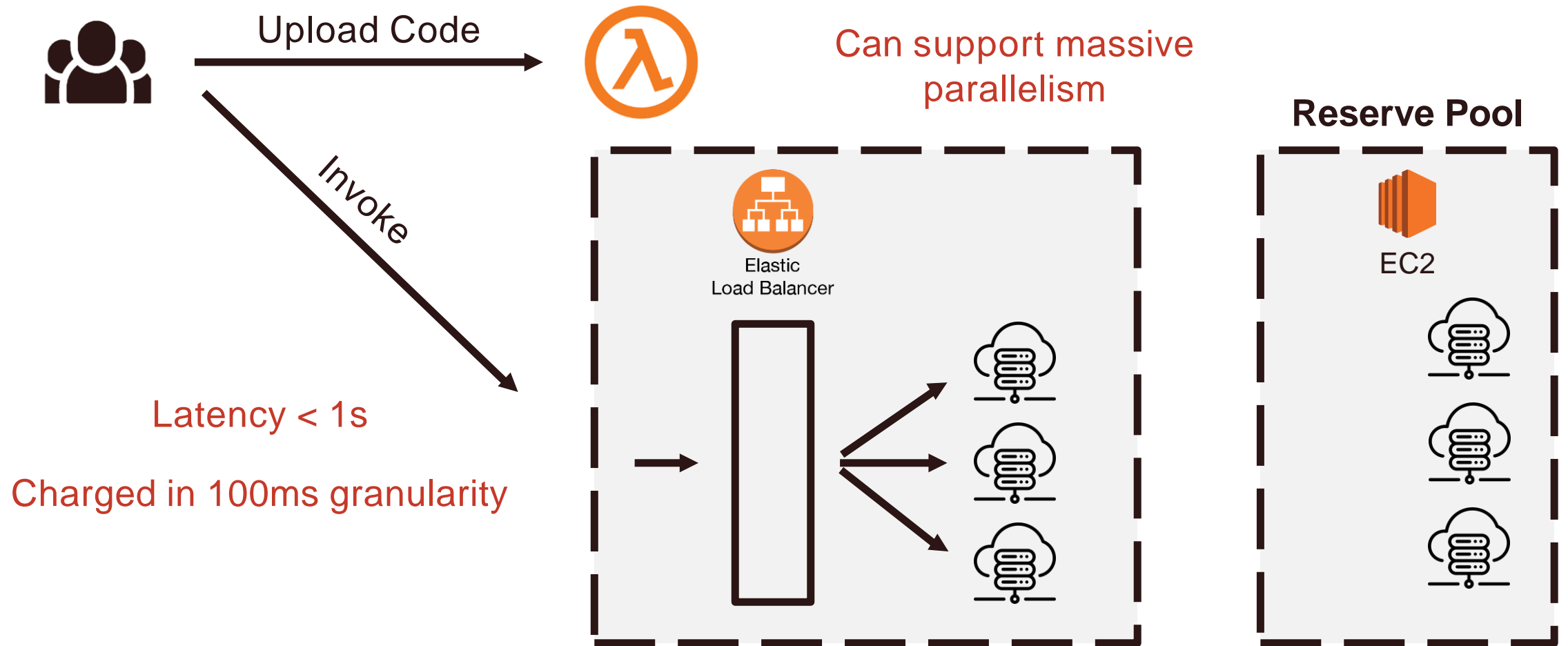
AWS Lambda



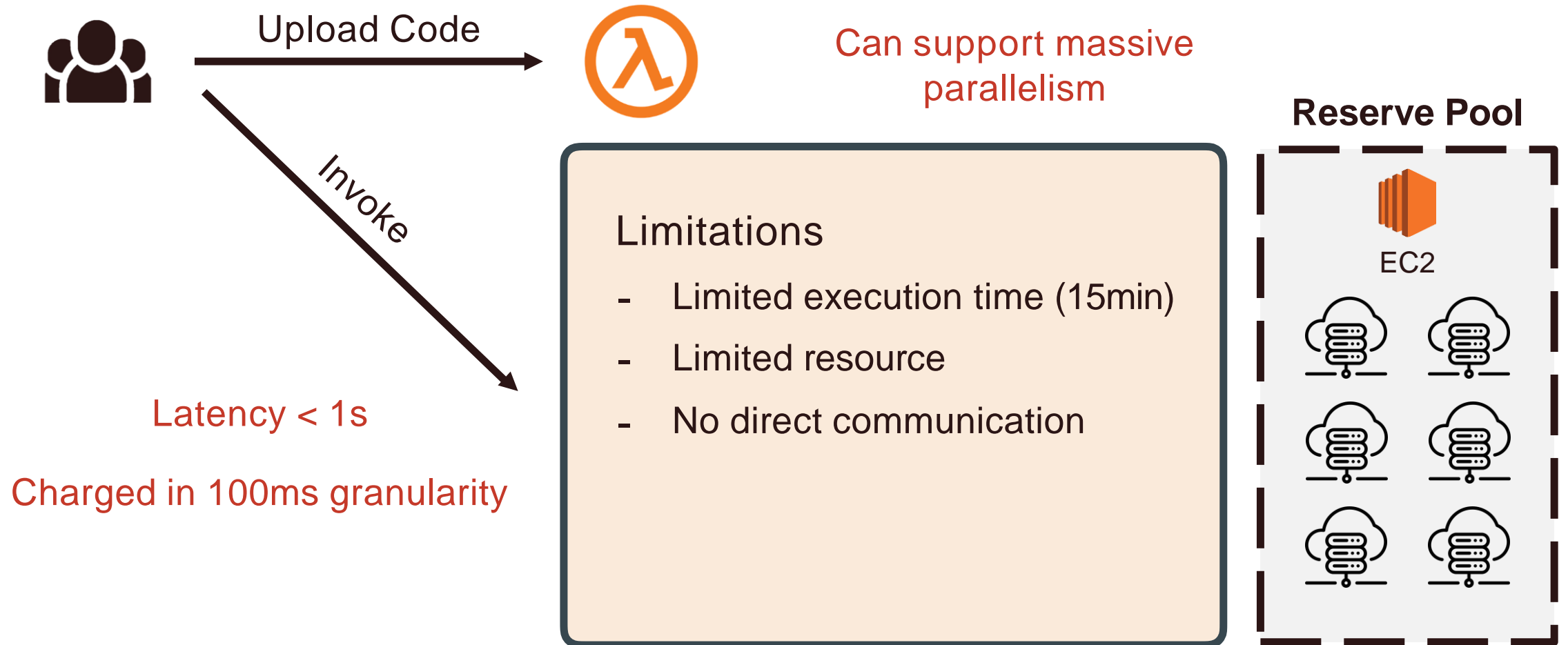
AWS Lambda



AWS Lambda



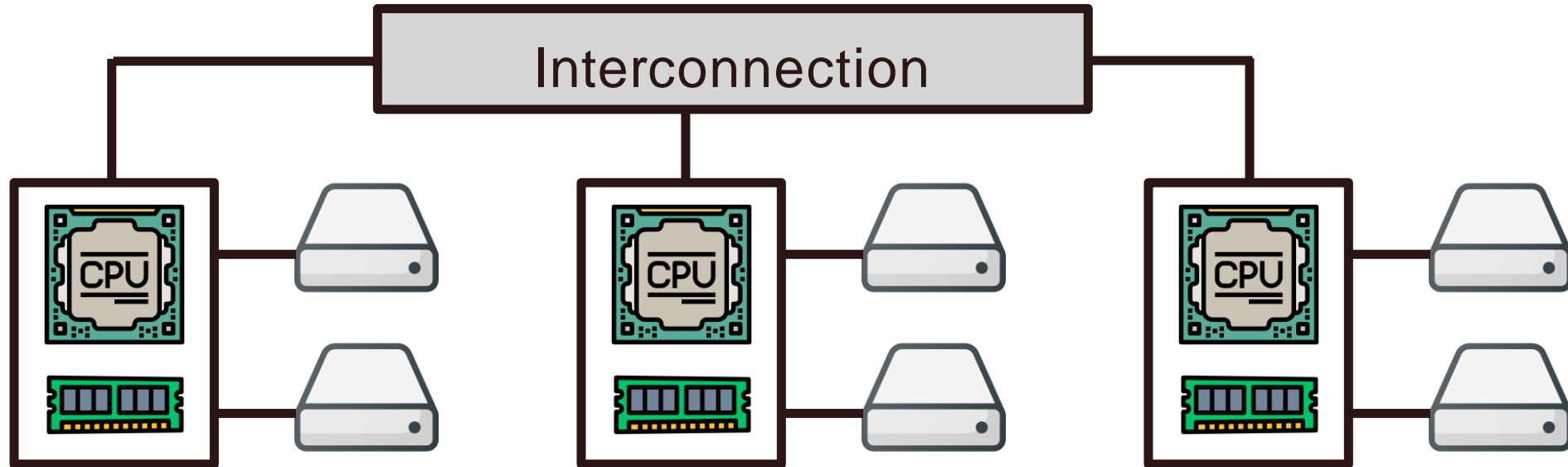
AWS Lambda



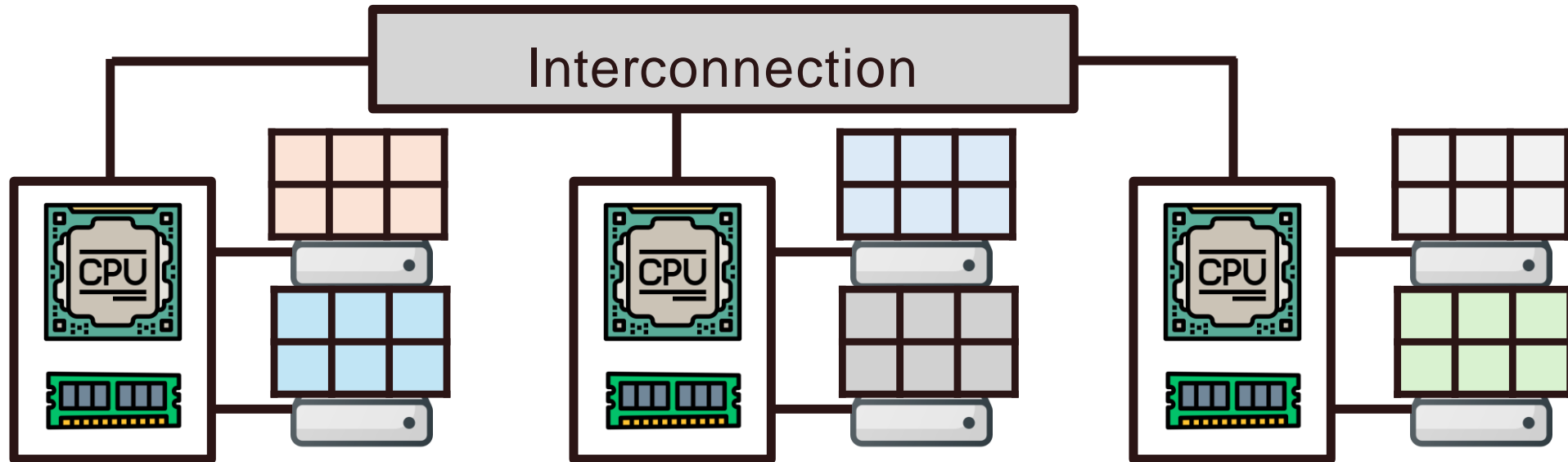
Cloud databases: outline

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

Shared-nothing architecture



Shared-nothing architecture



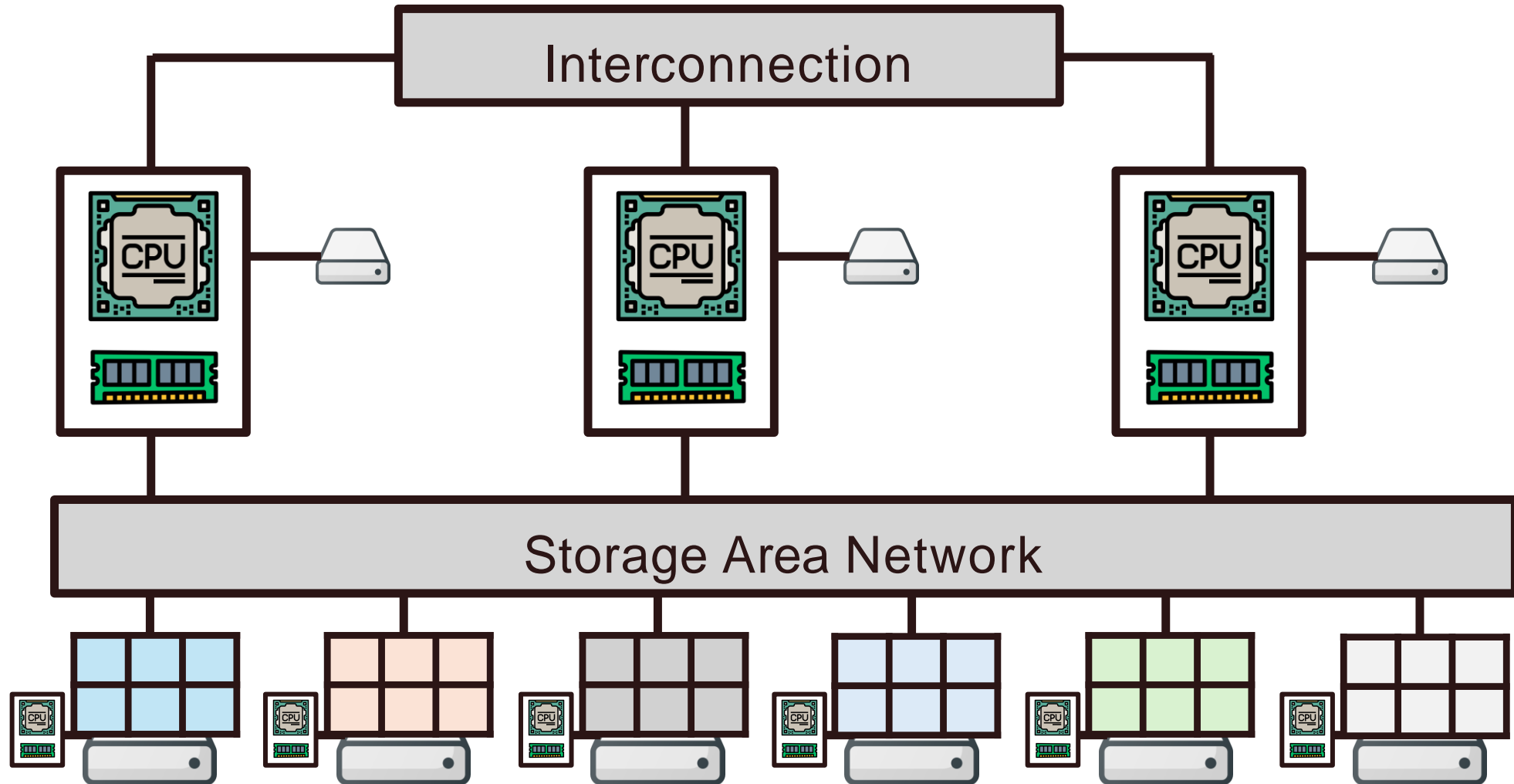
PROS

- Horizontal Scalability
- Simple and elegant design

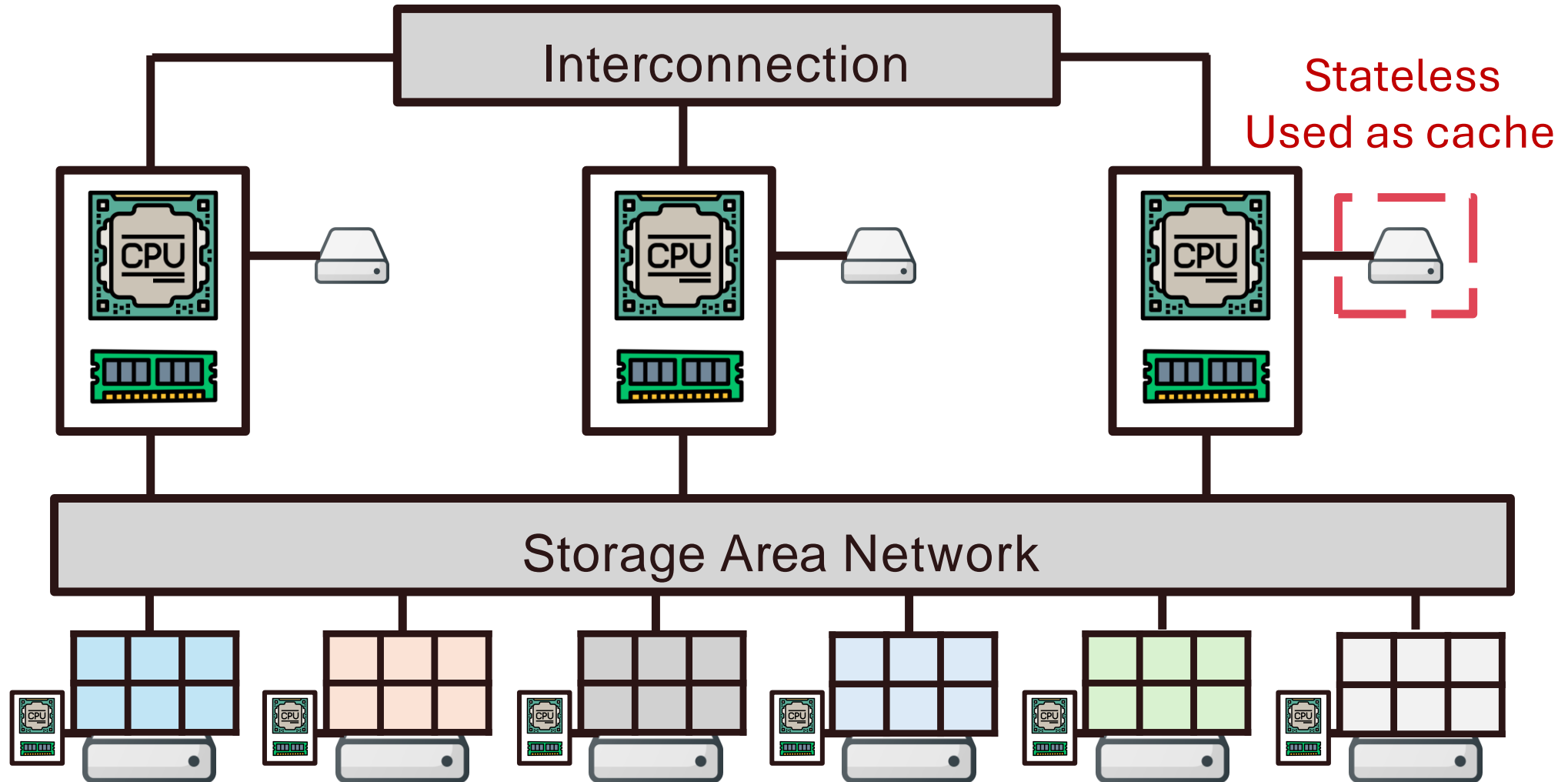
CONS

- Cross-machine operations
- Data redistribution at cluster resizing

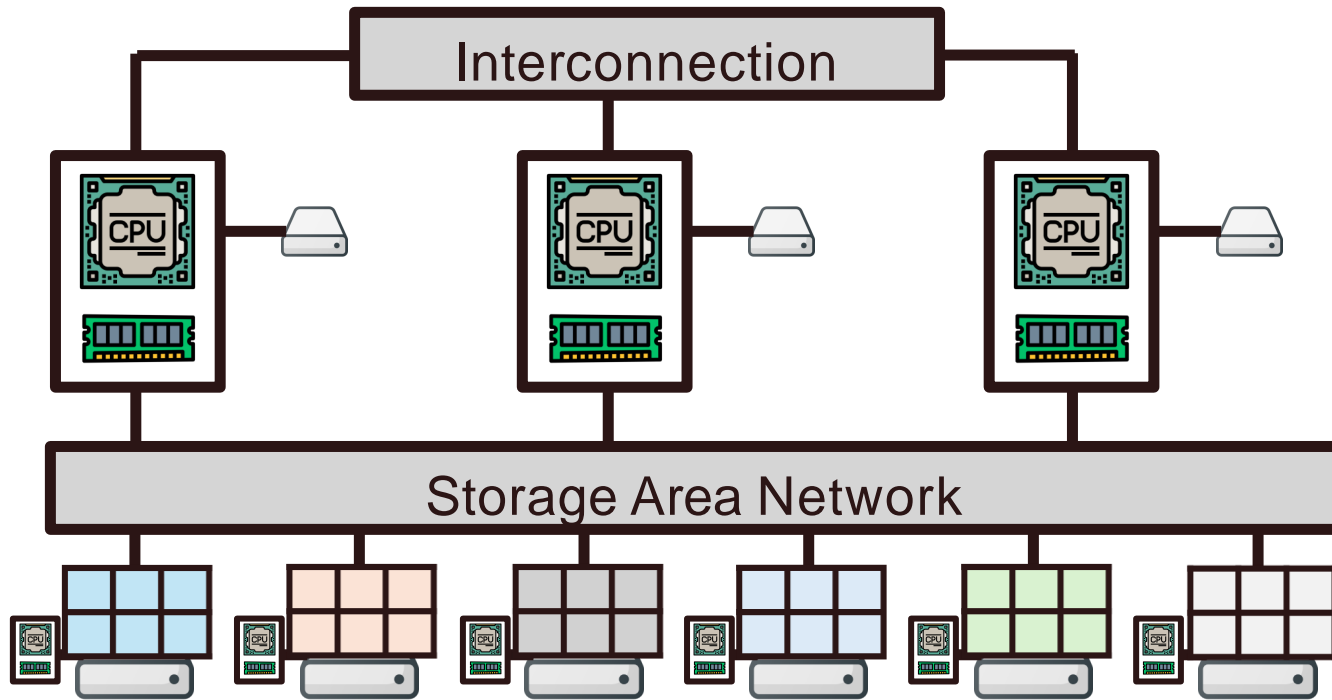
Separating compute and storage nodes



Separating compute and storage nodes



Separating compute and storage nodes

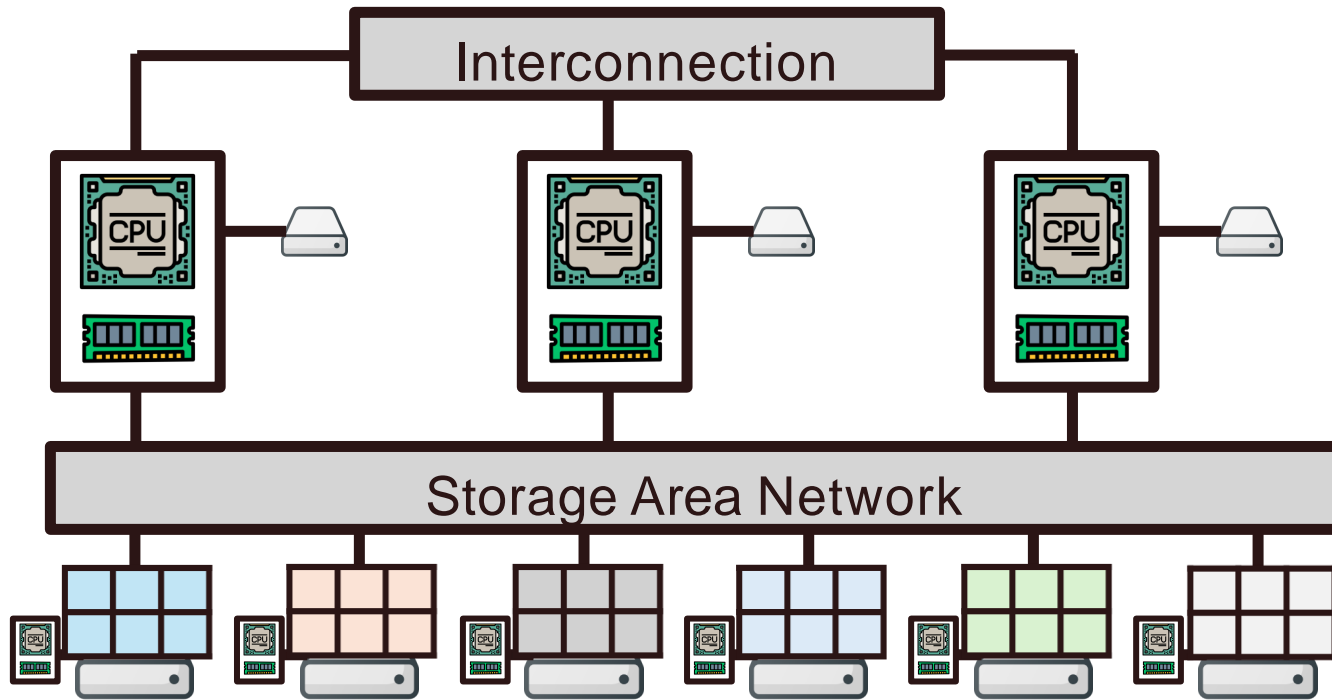


Key Benefit:

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 ?

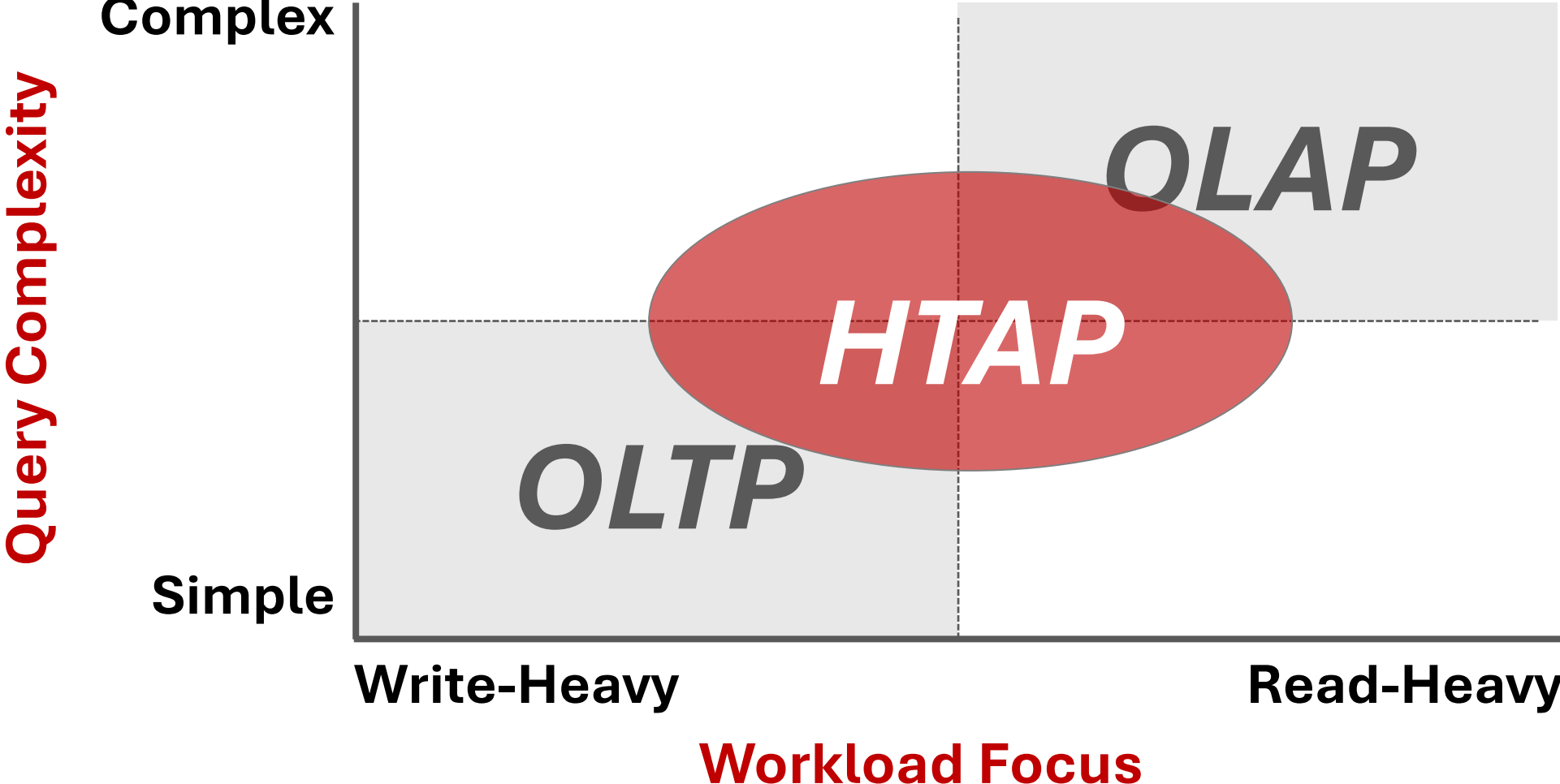
Cloud databases: outline

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

Database workloads

- Online Transaction Processing (OLTP)
 - Transactions that read/update a small amount of data each time.
 - Each transaction finishes in a short time (e.g., 1 or 0.1 milliseconds)
- Online Analytical Processing (OLAP)
 - Complex queries that read a lot of data to compute joins/aggregates.
 - A query can take up to hours or days.
 - Data is typically either unchanged or insert-only.
 - A query typically does not take locks. The DBMS uses a special way to maintain consistency if there are inserts.
- Hybrid Transaction + Analytical Processing
 - OLTP + OLAP together on the same database instance

Database workloads



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 = ...
```

Large number of concurrent operations

Online Analytical Processing (OLAP)

Example: data analytics, business report, ...

```
with v1 as(
  select i_category, i_brand, cc_name, d_year, d_moy,
         sum(cs_sales_price) sum_sales,
         avg(sum(cs_sales_price)) over
           (partition by i_category, i_brand,
                        cc_name, d_year)
         avg_monthly_sales,
         rank() over
           (partition by i_category, i_brand,
                        cc_name
            order by d_year, d_moy) rn
  from item, catalog_sales, date_dim, call_center
  where cs_item_sk = i_item_sk and
        cs_sold_date_sk = d_date_sk and
        cc_call_center_sk= cs_call_center_sk and
        (
          d_year = 1999 or
          ( d_year = 1999-1 and d_moy =12) or
          ( d_year = 1999+1 and d_moy =1)
        )
  group by i_category, i_brand,
           cc_name , d_year, d_moy),
v2 as(
  select v1.i_category ,v1.d_year, v1.d_moy ,v1.avg_monthly_sales
         ,v1.sum_sales, v1_lag.sum_sales psum, v1_lead.sum_sales nsum
  from v1, v1 v1_lag, v1 v1_lead
  where v1.i_category = v1_lag.i_category and
        v1.i_category = v1_lead.i_category and
        v1.i_brand = v1_lag.i_brand and
        v1.i_brand = v1_lead.i_brand and
        v1.cc_name = v1_lag.cc_name and
        v1.cc_name = v1_lead.cc_name and
        v1.rn = v1_lag.rn + 1 and
        v1.rn = v1_lead.rn - 1)
select *
from v2
where d_year = 1999 and
      avg_monthly_sales > 0 and
      case when avg_monthly_sales > 0 then abs(sum_sales - avg_monthly_sales) / avg_monthly_sales > 3
order by sum_sales - avg_monthly_sales, 3
limit 100;
```

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

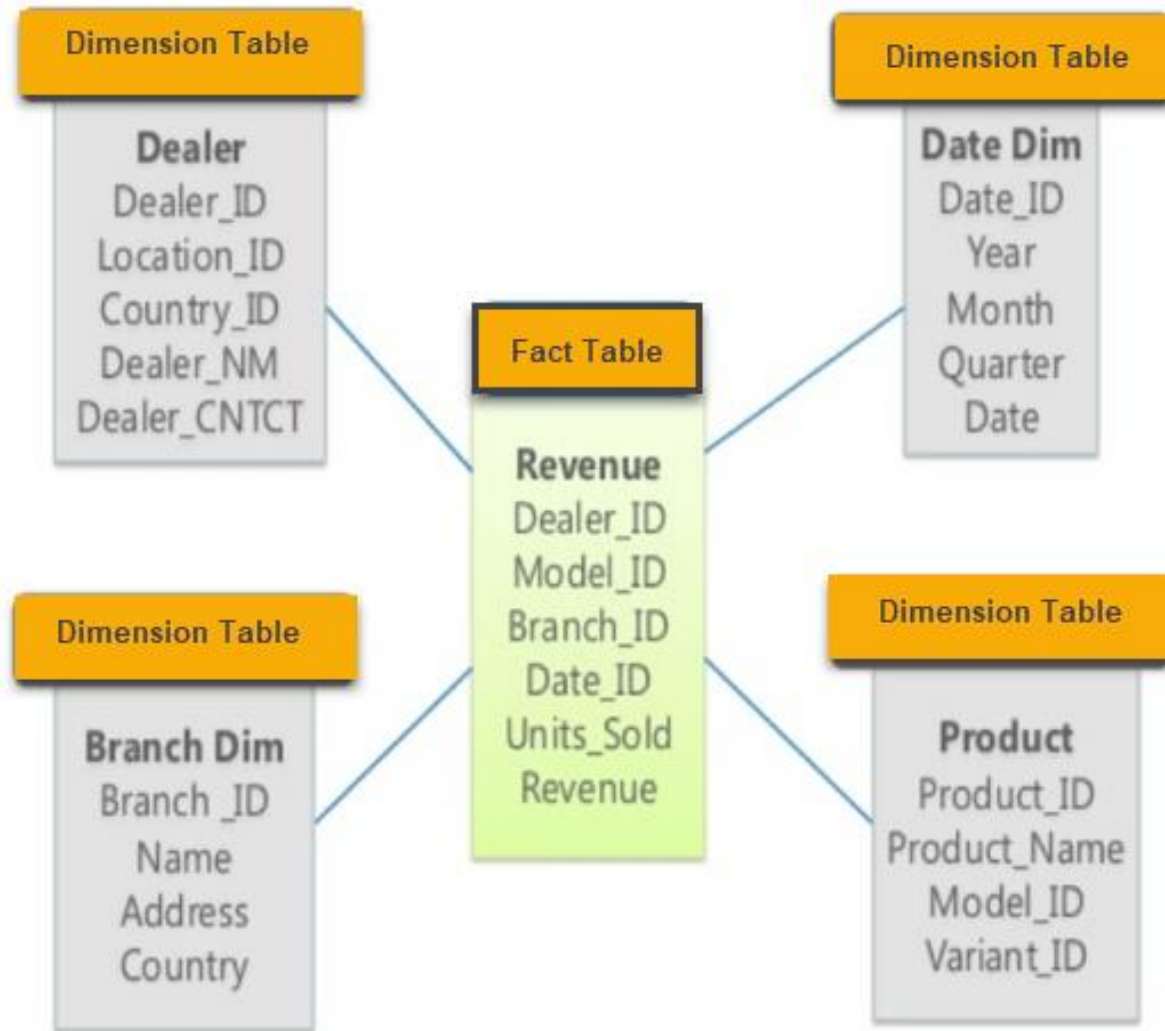
Heavy compute on large volume of data

```
with v1 as(
  select i_category, i_brand, cc_name, d_year, d_moy,
         sum(cs_sales_price) sum_sales,
         avg(sum(cs_sales_price)) over
           (partition by i_category, i_brand,
                        cc_name, d_year)
         avg_monthly_sales,
         rank() over
           (partition by i_category, i_brand,
                        cc_name
            order by d_year, d_moy) rn
  from item, catalog_sales, date_dim, call_center
  where cs_item_sk = i_item_sk and
        cs_sold_date_sk = d_date_sk and
        cc_call_center_sk= cs_call_center_sk and
        (
          d_year = 1999 or
          ( d_year = 1999-1 and d_moy =12) or
          ( d_year = 1999+1 and d_moy =1)
        )
  group by i_category, i_brand,
           cc_name , d_year, d_moy),
v2 as(
  select v1.i_category ,v1.d_year, v1.d_moy ,v1.avg_monthly_sales
        ,v1.sum_sales, v1_lag.sum_sales psum, v1_lead.sum_sales nsum
  from v1, v1 v1_lag, v1 v1_lead
  where v1.i_category = v1_lag.i_category and
        v1.i_category = v1_lead.i_category and
        v1.i_brand = v1_lag.i_brand and
        v1.i_brand = v1_lead.i_brand and
        v1.cc_name = v1_lag.cc_name and
        v1.cc_name = v1_lead.cc_name and
        v1.rn = v1_lag.rn + 1 and
        v1.rn = v1_lead.rn - 1)
select *
from v2
where d_year = 1999 and
      avg_monthly_sales > 0 and
      case when avg_monthly_sales > 0 then abs(sum_sales - avg_monthly_sales) / avg_monthly_sales > 0.3
order by sum_sales - avg_monthly_sales, 3
limit 100;
```

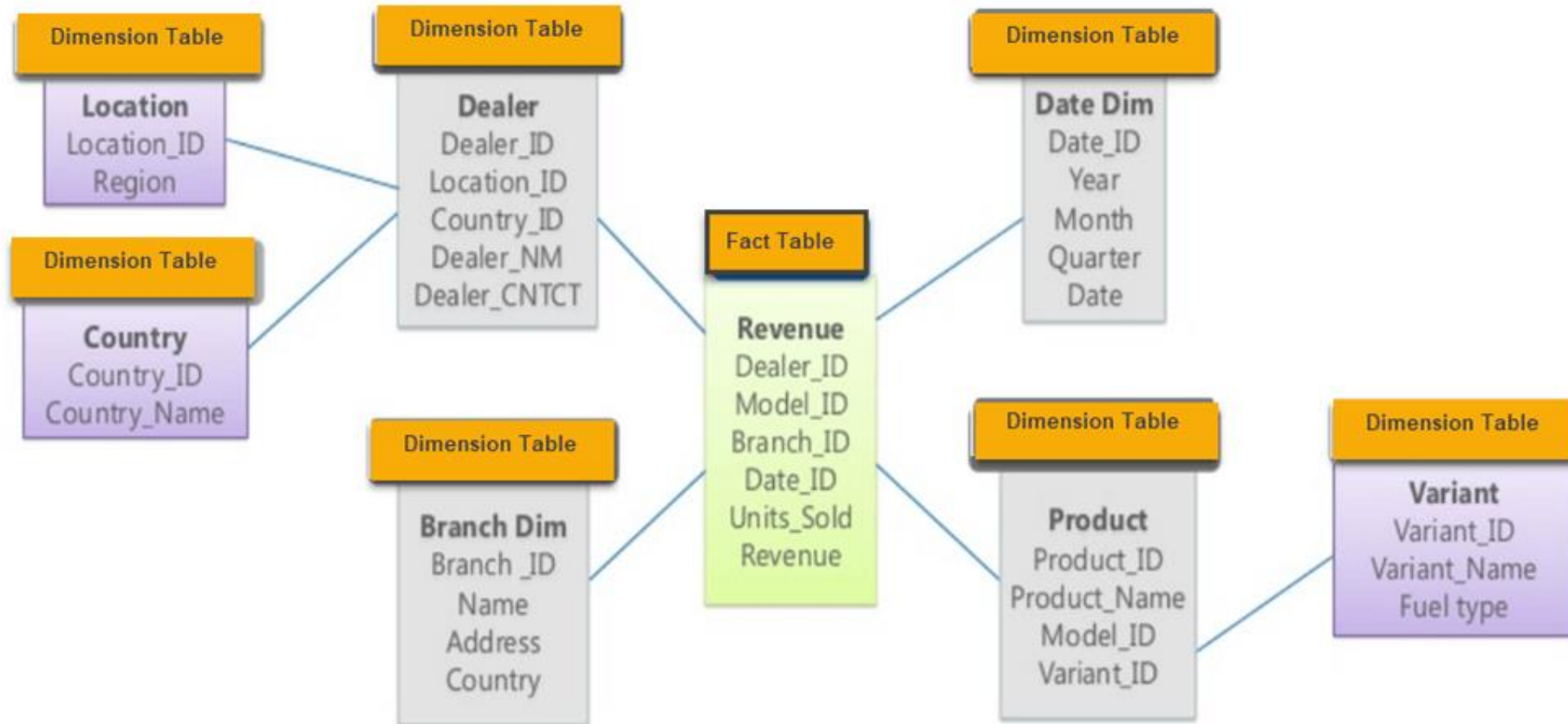
OLAP Schema

- An OLAP database is typically composed of fact tables and dimension tables.
- Fact tables record information about individual events, such as sales, and are usually very large.
 - Example: sales information for a retail store, with one tuple for each item that is sold.
- Dimension tables includes the attributes for describing the data in the fact table.
 - Example: Time and location for each item that is sold
- Fact tables and dimension tables are connected via foreign keys.

Star schema



Snowflake schema



Observation

- The way we store data will significantly impact the performance of processing queries.
- The relational model does not specify that the DBMS must store all of a tuple's attributes together on a single page.
- This may not be the best layout for OLAP workloads...

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

Pros

- Fast tuple insertion/deletion
- Fast SELECT *

Ideal for OLTP

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

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

Hybrid columnar format

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

Pure Columnar

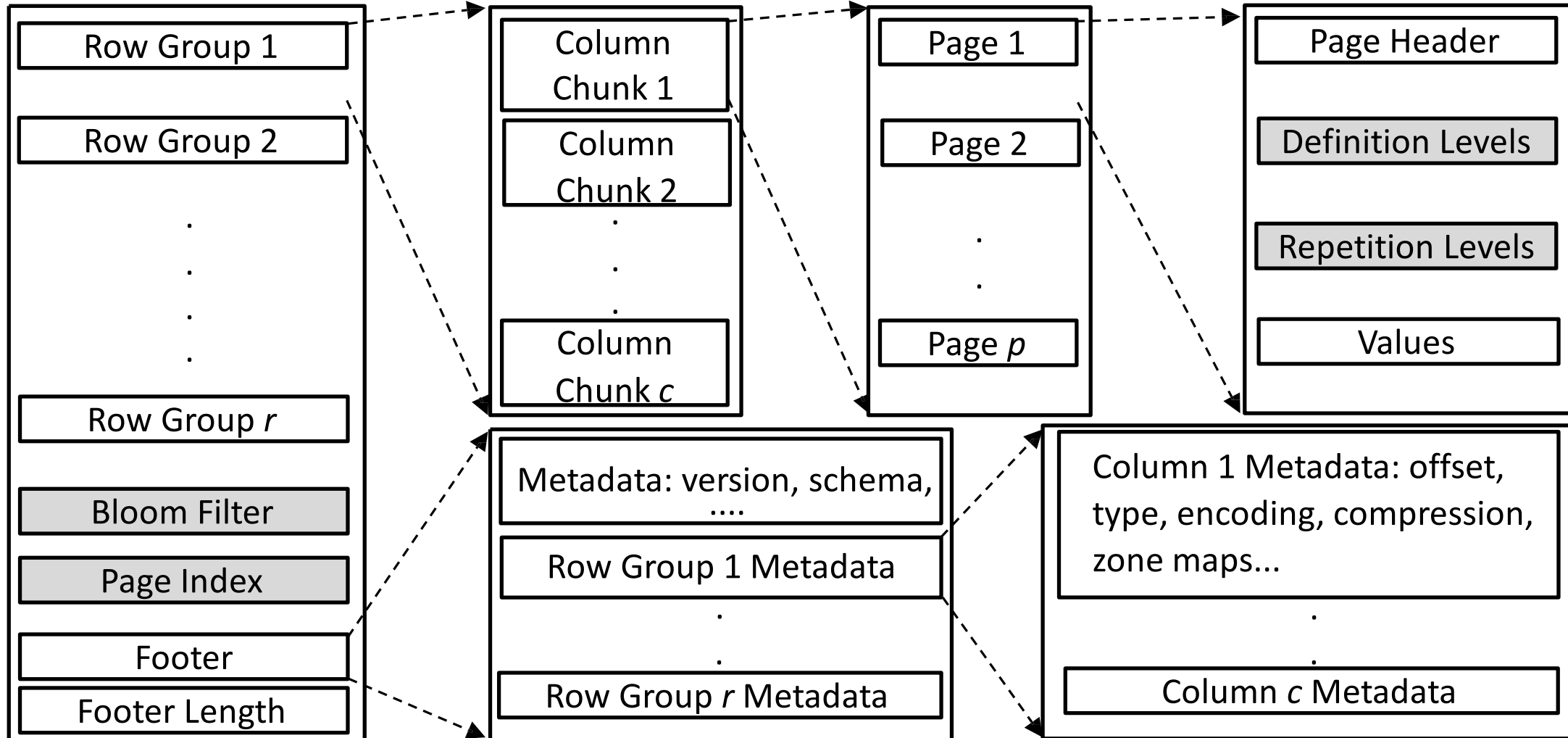
File 1	File 2	File 3
name	age	balance
Alice	18	1000
David	18	1500
Bob	23	2000
Emily	20	2500
Bob	18	3000
Emily	22	3500
Bob	19	4000
Charlie	20	4500
Emily	19	5000

Hybrid Columnar (aka., PAX)

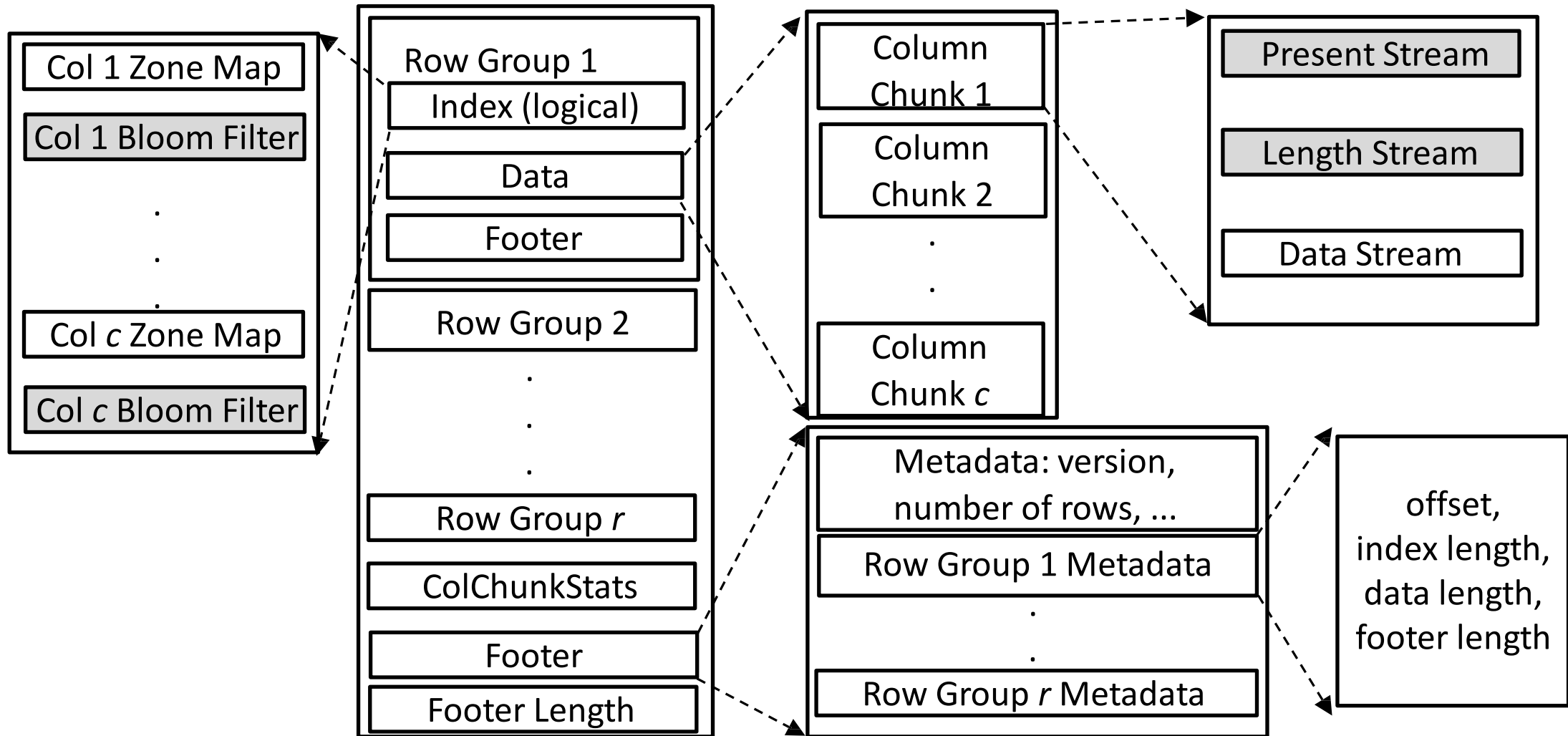
File 1	name	age	balance
	Alice	18	1000
	David	18	1500
	Bob	23	2000
File 2	name	age	balance
	Emily	20	2500
	Bob	18	3000
	Emily	22	3500
File 3	name	age	balance
	Bob	19	4000
	Charlie	20	4500
	Emily	19	5000

A disk page in the original proposal

Apache Parquet



Apache ORC



Columnar compression

- Dictionary Encoding
- Run-length Encoding
- Bit-Packing Encoding
- Bitmap Encoding
- Delta Encoding
- Learned encoding

Naïve Compression

Compress data using a general-purpose algorithm. The scope of compression is only based on the data provided as input.

→ [LZO](#) (1996), [LZ4](#) (2011), [Snappy](#) (2011), [Oracle OZIP](#) (2014), [Zstd](#) (2015)

Considerations

- Computational overhead
- Compress vs. decompress speed.

Naïve compression

The DBMS must decompress data first before it can be read and (potentially) modified.

→ Repeated compression and decompression will be the bottleneck.

These schemes also do not consider the high-level meaning or semantics of the data.

Dictionary encoding

Replace frequent values with smaller fixed-length codes and then maintain a mapping (dictionary) from the codes to the original values

- Typically, one code per attribute value.
- Most widely used native compression scheme in DBMSs.

The ideal dictionary scheme supports fast encoding and decoding for both point and range queries.

Dictionary encoding example

```
SELECT * FROM users  
WHERE name = 'Andy'
```



```
SELECT * FROM users  
WHERE name = 30
```

Original Data

name
Andrea
Prashanth
Andy
Matt
Prashanth



Compressed Data

name
10
20
30
40
20

value	code
Andrea	10
Prashanth	20
Andy	30
Matt	40

Dictionary

Dictionary encoding: order preserving

The encoded values need to support the same collation as the original values.

```
SELECT * FROM users
WHERE name LIKE 'And%'
```



```
SELECT * FROM users
WHERE name BETWEEN 10 AND 20
```

Original Data

name
Andrea
Prashanth
Andy
Matt
Prashanth



Compressed Data

<i>name</i>	<i>value</i>	<i>code</i>
10	Andrea	10
40	Andy	20
20	Matt	30
30	Prashanth	40
40		

Sorted
Dictionary

Dictionary: order preserving

```
SELECT name FROM users  
WHERE name LIKE 'And%'
```



Still must perform scan on column

```
SELECT DISTINCT name  
FROM users  
WHERE name LIKE 'And%'
```



Only need to access dictionary

Original Data

name
Andrea
Prashanth
Andy
Matt
Prashanth



Compressed Data

name	value	code
10	Andrea	10
40	Andy	20
20	Matt	30
30	Prashanth	40
40		

Sorted
Dictionary

Run-length encoding

Compress the same continuous values in a single column into triplets:

- The value of the attribute.
- The start position in the column segment.
- The # of elements in the run.

Requires the columns to be sorted intelligently to maximize compression opportunities.

Run-length encoding

Original Data

id	isDead
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y



Compressed Data

id	isDead
1	(Y,0,3)
2	(N,3,1)
3	(Y,4,1)
4	(N,5,1)
6	(Y,6,2)
7	
8	
9	

RLE Triplet
- Value
- Offset
- Length

Run-length encoding

Sorted Data

id	isDead
1	Y
2	Y
3	Y
6	Y
8	Y
9	Y
4	N
7	N



Compressed Data

id	isDead
1	(Y,0,6)
2	(N,7,2)
3	
6	
8	
9	
4	
7	

RLE Triplet
- Value
- Offset
- Length

Bit packing

If the values for an integer attribute is smaller than the range of its given data type size, we can reduce the number of bits to represent each value.

Use bit-shifting tricks to operate on multiple values in a single word.

Original Data

int32
13
191
56
92
81
120
231
172

*Original:
8 × 32-bits =
256 bits*

00000000 00000000 00000000 00001101
00000000 00000000 00000000 10111111
00000000 00000000 00000000 00111000
00000000 00000000 00000000 01011100
00000000 00000000 00000000 01010001
00000000 00000000 00000000 01111000
00000000 00000000 00000000 11100111
00000000 00000000 00000000 10101100

*Compressed:
8 × 8-bits =
64 bits*

Patching/mostly encoding

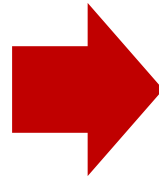
A variation of bit packing when attribute's values are "mostly" less than the largest size.

→ The remaining values that cannot be compressed are stored in their raw form.

Original:
 $8 \times 32\text{-bits} =$
 256 bits

Original Data

int32
13
191
99999999
92
81
120
231
172



Compressed Data

mostly8	offset	value
13	3	99999999
181		
XXX		Invalid
92		
81		
120		
231		
172		

Compressed:
 $(8 \times 8\text{-bits}) +$
 $16\text{-bits} + 32\text{-bits}$
 $= 112\text{ bits}$

Bitmap encoding

Store a separate bitmap for each unique value for an attribute

- The i^{th} position in the Bitmap corresponds to the i^{th} tuple in the table.
- Each bit indicates whether the attribute of the i^{th} tuple is this value.
- Typically segmented into chunks to avoid allocating large blocks of contiguous memory.

Only practical if the number of distinct values is low.

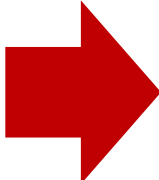
Some DBMSs provide bitmap indexes.

Bitmap encoding example

Original Data

id	isDead
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Original:
 $9 \times 8\text{-bits} = 72\text{ bits}$



Compressed:
 $16\text{ bits} + 18\text{ bits} = 34\text{ bits}$

Compressed Data

id	isDead	
	Y	N
1	1	0
2	1	0
3	1	0
4	0	1
6	1	0
7	0	1
8	1	0
9	1	0

$2 \times 8\text{-bits} = 16\text{ bits}$

$9 \times 2\text{-bits} = 18\text{ bits}$

Bitmap encoding example

Assume we have 10 million tuples.
43,000 zip codes in the US.

→ **10000000 × 32-bits = 40 MB**

→ **10000000 × 43000 = 53.75 GB**

Every time the application inserts a new tuple, the DBMS must extend 43,000 different bitmaps.

```
CREATE TABLE customer (  
  id INT PRIMARY KEY,  
  name VARCHAR(32),  
  email VARCHAR(64),  
  address VARCHAR(64),  
  zip_code INT  
);
```

Delta encoding

Recording the difference between values that follow each other in the same column.

- Store base value in-line or in a separate look-up table.
- Combine with RLE to get even better compression ratios.

Original Data

time64	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

**$5 \times 64\text{-bits}$
 $= 320\text{ bits}$**



Compressed Data

time64	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

**$64\text{-bits} + (4 \times 16\text{-bits})$
 $= 128\text{ bits}$**



Compressed Data

time64	temp
12:00	99.5
(+1,4)	-0.1
	+0.1
	+0.1
	-0.2

**$64\text{-bits} + (2 \times 16\text{-bits})$
 $= 96\text{ bits}$**

Value sequence compression

Value Sequence: $V_1, V_2, V_3, \dots, V_n$

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

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

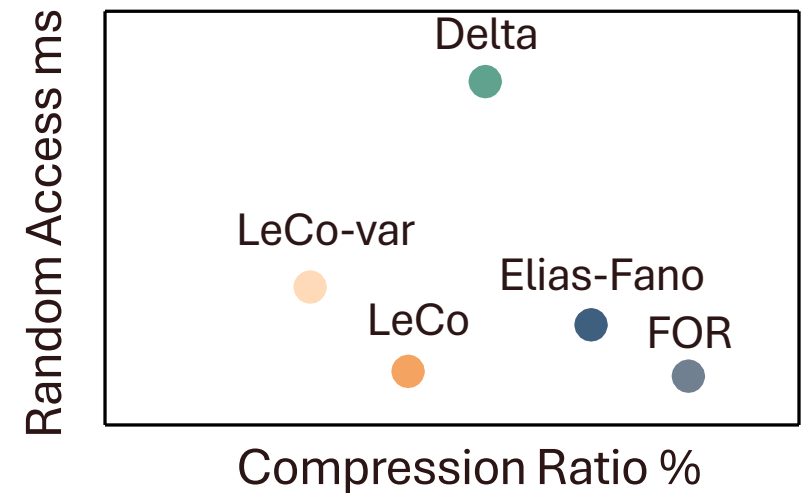
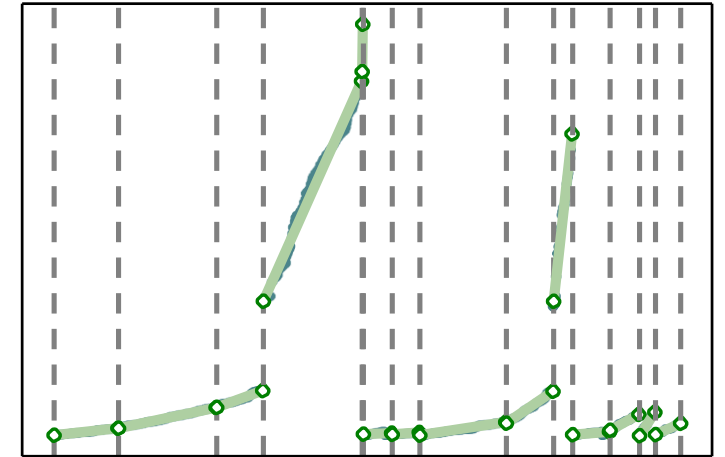
Values $\vec{v}_{[0,n)} = (v_0, \dots, v_{n-1})$

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

Models F_0 F_1 F_{m-1}

Deltas $\Delta_i = v_i - F_j(v_i), \quad \text{for } v_i \in \vec{v}_{[k_j,k_{j+1})}$

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



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.
- 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!

Conclusion

- It is important to choose the right storage model for the target workload:
 - OLTP = Row Store
 - OLAP = Column Store
- OLAP databases rely on columnar compression for high compression speed and compression ratio.
- Compressed data can be directly used to support some queries without decompression.

Cloud databases: outline

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

Data storage

In distributed data analysis, data is typically distributed across nodes.

- Option 1: Data is partitioned and co-located with compute nodes.
 - This is the shared-nothing architecture we discussed earlier.
 - Queries are typically pushed to the nodes where the data is.
- Option 2: Data is partitioned and stored in a dedicated storage layer.
 - This is more modern architecture in the cloud.
 - Data is pulled from the storage layer to the compute layer.
 - Optimization: retrieving data from the storage layer with basic filtering capability.
 - S3 select supports retrieving data from CSV/JSON/Parquet.

Storage models

Choice #1: N-ary Storage Model (NSM)

- This is for OLTP and what we learned before.

Choice #2: Decomposition Storage Model (DSM)

- This is for OLAP

Choice #3: Hybrid Storage Model (PAX)

- This is for HTAP (Hybrid Transactional and Analytical Processing) and OLAP workloads.

N-Ary Storage Model (NSM)

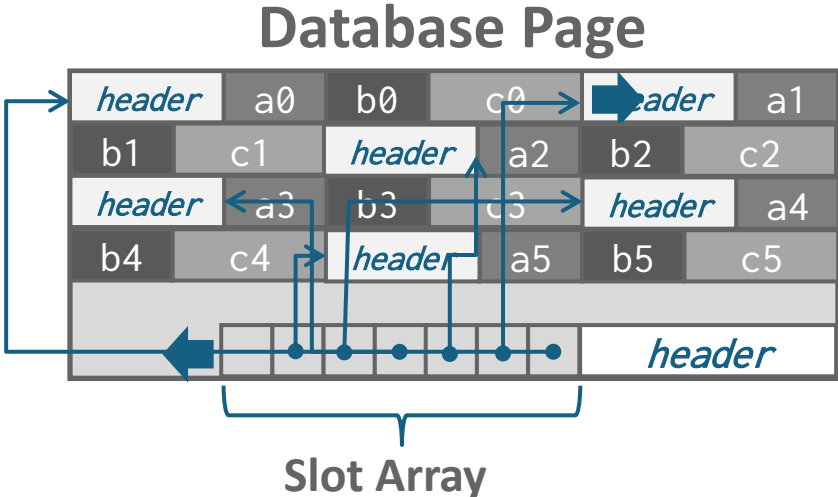
- The DBMS stores (almost) all attributes for a single tuple contiguously on a single page.
 - It is also known as a “row store.”
- Ideal for OLTP workloads where queries are more likely to access a small number of individual rows and execute write-heavy workloads.
- NSM database page sizes are typically some constant multiple of 4 KB hardware pages.
 - Oracle (4 KB), Postgres (8 KB), MySQL (16 KB)

NSM: Physical Organization

A disk-oriented NSM system stores a tuple's fixed-length and variable-length attributes contiguously in a single-slotted page.

The tuple's record id (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

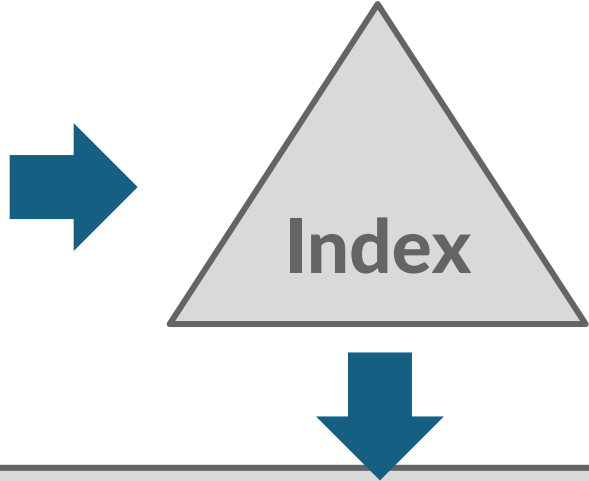
	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



NSM: OLTP example

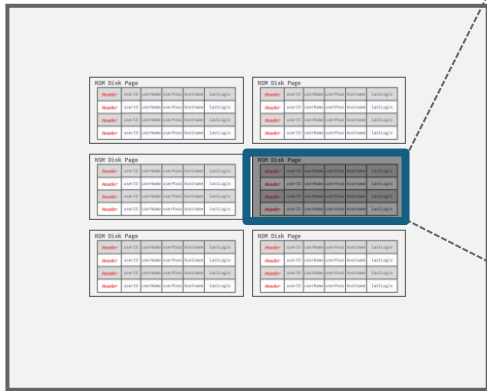
```
SELECT * FROM useracct  
WHERE userName = ?  
AND userPass = ?
```

```
INSERT INTO useracct  
VALUES (?, ?, ...?)
```



Disk

Database File



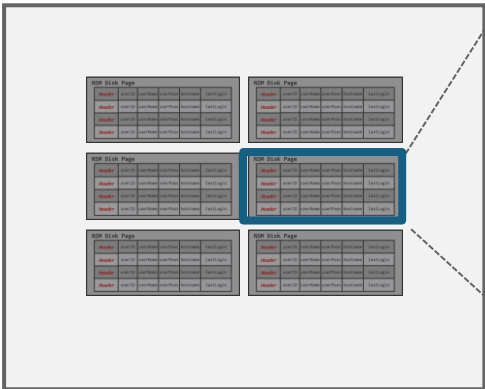
<i>NSM Disk Page</i>					
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin

NSM: OLAP example

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
FROM useracct AS U  
WHERE U.hostname LIKE '%.gov'  
GROUP BY EXTRACT(month FROM U.lastLogin)
```



Database File



NSM Disk Page

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin

Useless Data

NSM: summary

Advantages

- Fast inserts, updates, and deletes for single rows.
- Good for queries that need the entire tuple (OLTP).
- Can use index-oriented physical storage for clustering.

Disadvantages

- Not good for scanning large portions of the table and/or a subset of the attributes.
- Terrible memory locality for OLAP access patterns.
- Not ideal for compression because of multiple value domains within a single page.

Decomposition Storage Model (DSM)

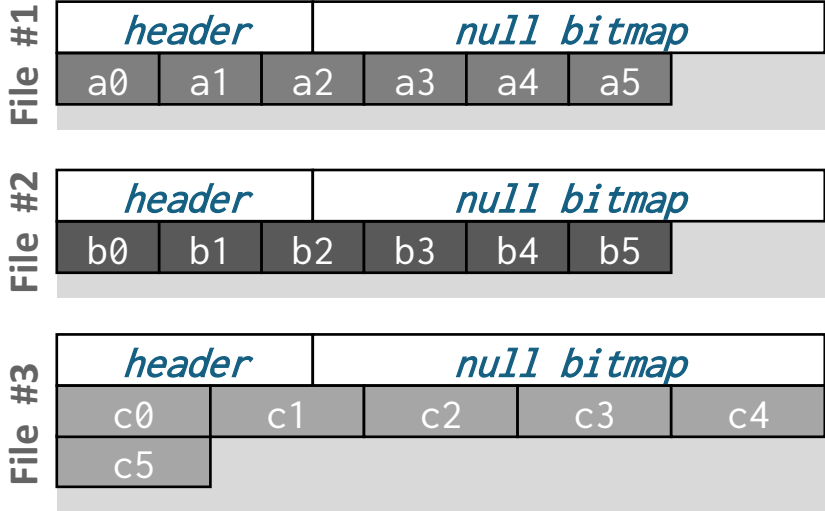
- The DBMS stores a single attribute of all tuples contiguously in a block of data.
→ “column store” mentioned earlier.
- Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table’s attributes.
- DBMS is responsible for combining/splitting a tuple’s attributes when reading/writing.

DSM: Physical Organization

- Store each attribute and its metadata (e.g., nulls) in a separate array.
 - Most systems construct physical tuples using offsets into these arrays if an array includes **fixed-length** data.
 - Need to handle variable-length values...

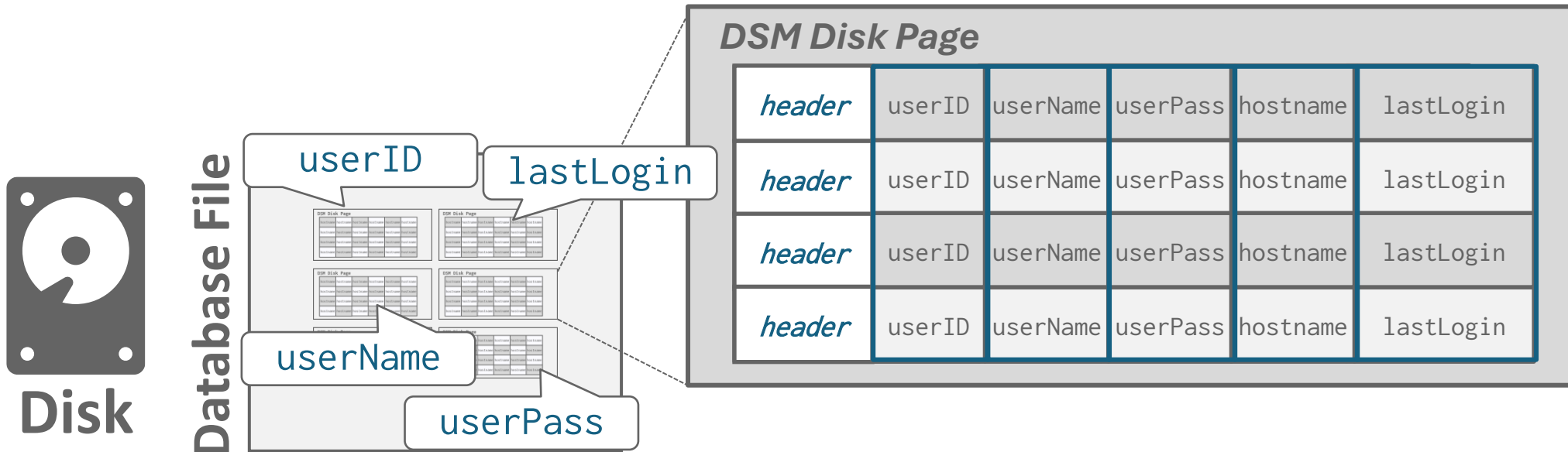
	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5

- Each array is stored as a separate file with a dedicated header area for metadata about the entire column.



DSM: database example

The DBMS stores the values of a single attribute across multiple tuples contiguously in a page.



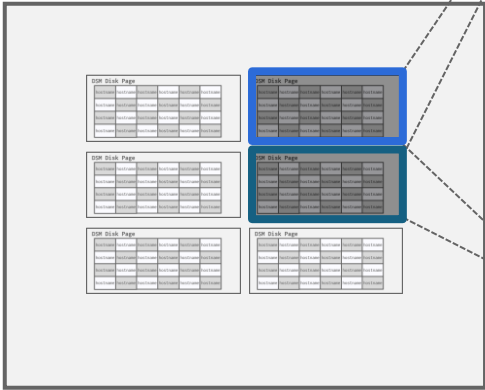
DSM: database example

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
FROM useracct AS U  
WHERE U.hostname LIKE '%.gov'  
GROUP BY EXTRACT(month FROM U.lastLogin)
```



Disk

Database File



DSM Disk Page

<i>header</i>	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	lastLogin

DSM: tuple construction

Choice #1: Fixed-length Offsets

- Each value is the same length for an attribute.
- To find the *i*th tuple, we could find the *i*th value for each column using the offset.

Choice #2: Embedded Tuple Ids

- Each value is stored with its tuple id in a column.

Offsets

	A	B	C	D
0				
1				
2				
3				

Embedded Ids

	A	B	C	D
0		1	1	3
1		0	1	2
2		2	0	1
3		3	3	0

Tradeoffs

- Fixed-length Offsets
 - + No metadata is required for constructing tuples.
 - + Fast tuple construction.
 - - Only works for fixed-length attributes.
 - - The positions of the arrays for separate attributes should be the same.
- Embedded Tuple Ids
 - + Works for both fixed-length and variable-length attributes.
 - + The positions of the arrays for separate attributes could be different.
 - This enables more compression opportunities.
 - - Additional metadata (i.e., tuple Ids) for constructing tuples.
 - - Tuple construction is slower.

DSM: Summary

Advantages

- Reduces the amount wasted I/O per query because the DBMS only reads the data that it needs.
- Faster query processing because of increased locality and cached data reuse.
- Better data compression (more on this in later slides).

Disadvantages

- Slow for point queries, inserts, updates, and deletes

Observation on DSM

OLAP queries rarely access a single column in a table by itself.

→ At some point during query execution, the DBMS must get other columns and stitch the original tuple back together.

But we still want to store data in a columnar format to get the storage + execution benefits.

So, we need a columnar scheme that stores attributes separately but keeps the data for each tuple physically close to each other...

PAX storage model

Partition Attributes Across (PAX) is a hybrid storage model that sits between NSM and DSM.

The goal is to get the benefit of faster processing on columnar storage while retaining the spatial locality benefits of row storage.

Weaving Relations for Cache Performance

Anastassia Ailamaki[‡]
Carnegie Mellon University
natassa@cs.cmu.edu

David J. DeWitt
Univ. of Wisconsin-Madison
dewitt@cs.wisc.edu

Mark D. Hill
Univ. of Wisconsin-Madison
markhill@cs.wisc.edu

Marios Skounakis
Univ. of Wisconsin-Madison
marios@cs.wisc.edu

Abstract

Relational database systems have traditionally optimized for I/O performance and organized records sequentially on disk pages using the *N*-ary Storage Model (NSM) (a.k.a., slotted pages). Recent research, however, indicates that cache utilization and performance is becoming increasingly important on modern platforms. In this paper, we first demonstrate that in-page data placement is the key to high cache performance and that NSM exhibits low cache utilization on modern platforms. Next, we propose a new data organization model called PAX (Partition Attributes Across), that significantly improves cache performance by grouping together all values of each attribute within each page. Because PAX only affects layout inside the pages, it incurs no storage penalty and does not affect I/O behavior. According to our experimental results, when compared to NSM (a) PAX exhibits superior cache and memory bandwidth utilization, saving at least 75% of NSM's stall time due to data cache accesses, (b) range selection queries and updates on memory-resident relations execute 17-25% faster, and (c) TPC-H queries involving I/O execute 11-48% faster.

1 Introduction

The communication between the CPU and the secondary storage (I/O) has been traditionally recognized as the major database performance bottleneck. To optimize data transfer to and from mass storage, relational DBMSs have long organized records in slotted disk pages using the *N*-ary Storage Model (NSM). NSM stores records contiguously starting from the beginning of each disk page, and uses an offset (slot) table at the end of the page to locate the beginning of each record [27].

Unfortunately, most queries use only a fraction of each record. To minimize unnecessary I/O, the Decomposition Storage Model (DSM) was proposed in 1985 [10]. DSM partitions an *n*-attribute relation vertically into *n* sub-relations, each of which is accessed only when the corresponding attribute is needed. Queries that involve multiple attributes from a relation, however, must spend

tremendous additional time to join the participating sub-relations together. Except for Sybase-IQ [33], today's relational DBMSs use NSM for general-purpose data placement [20][29][32].

Recent research has demonstrated that modern database workloads, such as decision support systems and spatial applications, are often bound by delays related to the processor and the memory subsystem rather than I/O [20][5][26]. When running commercial database systems on a modern processor, data requests that miss in the cache hierarchy (i.e., requests for data that are not found in any of the caches and are transferred from main memory) are a key memory bottleneck [1]. In addition, only a fraction of the data transferred to the cache is useful to the query: the item that the query processing algorithm requests and the transfer unit between the memory and the processor are typically not the same size. Loading the cache with useless data (a) wastes bandwidth, (b) pollutes the cache, and (c) possibly forces replacement of information that may be needed in the future, incurring even more delays. The challenge is to repair NSM's cache behavior without compromising its advantages over DSM.

This paper introduces and evaluates **Partition Attributes Across (PAX)**, a new layout for data records that combines the best of the two worlds and exhibits performance superior to both placement schemes by eliminating unnecessary accesses to main memory. For a given relation, PAX stores the same data on each page as NSM. *Within* each page, however, PAX groups all the values of a particular attribute together on a minipage. During a sequential scan (e.g., to apply a predicate on a fraction of the record), PAX fully utilizes the cache resources, because on each miss a number of a single attribute's values are loaded into the cache together. At the same time, all parts of the record are on the same page. To reconstruct a record one needs to perform a *mini-join* among minipages, which incurs minimal cost because it does not have to look beyond the page.

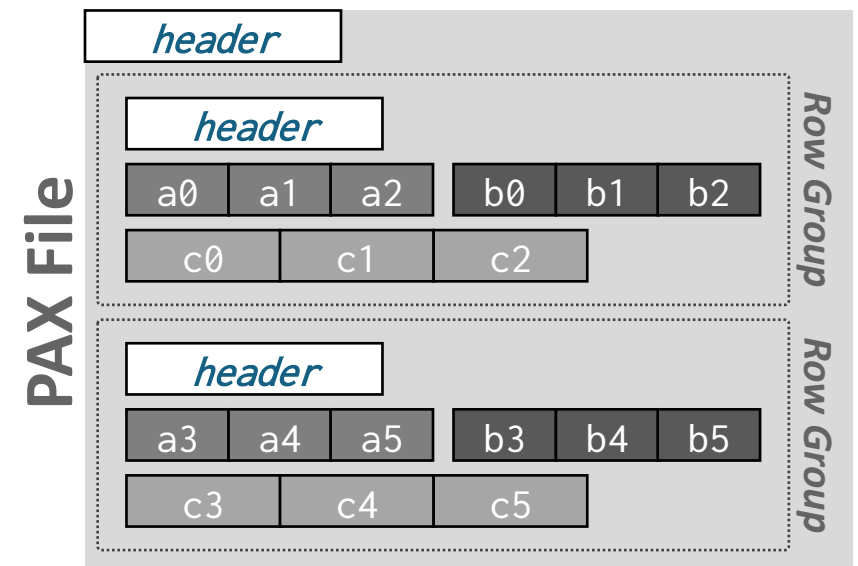
We evaluated PAX against NSM and DSM using (a) predicate selection queries on numeric data and (b) a variety of queries on TPC-H datasets on top of the Shore storage manager [7]. We vary query parameters including selectivity, projectivity, number of predicates, distance between the projected attribute and the attribute in the predicate, and degree of the relation. The experimental results show that, when compared to NSM, PAX (a) incurs 50-75% fewer second-level cache misses due to data

[‡] Work done while author was at the University of Wisconsin-Madison.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.
Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

PAX: physical organization

- It horizontally partitions rows into groups and vertically partitions their attributes into columns.
- Each row group contains its metadata header about its contents.
- PAX-style formats are widely adopted in big data processing
 - ORC (Optimized Row-Column File)
 - Parquet

	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5

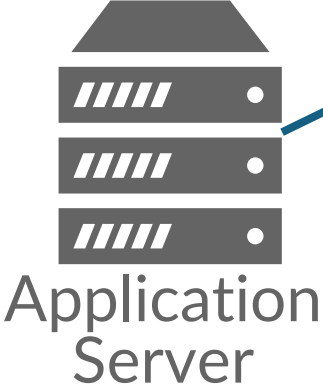


Observation on OLAP workloads

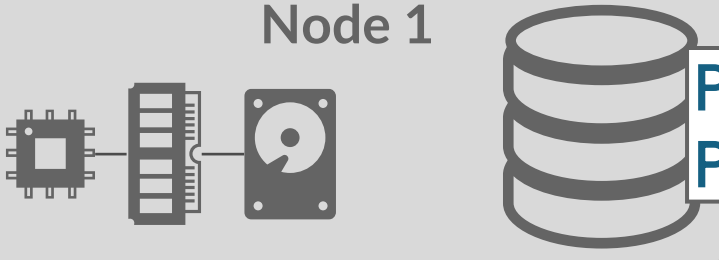
- OLAP requires accessing a lot of data, so I/O will be its main bottleneck during query execution.
- The DBMS can **compress** pages to reduce the data size and I/O operations.

Push query to data (shared nothing)

```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



Node 1

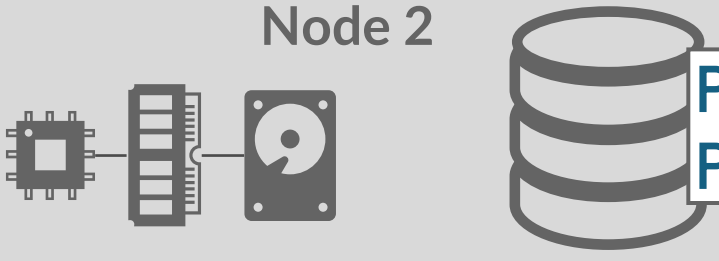


P1→R.id:1-100
P1→S.id:1-100

$R \bowtie S$
IDs [101,200]

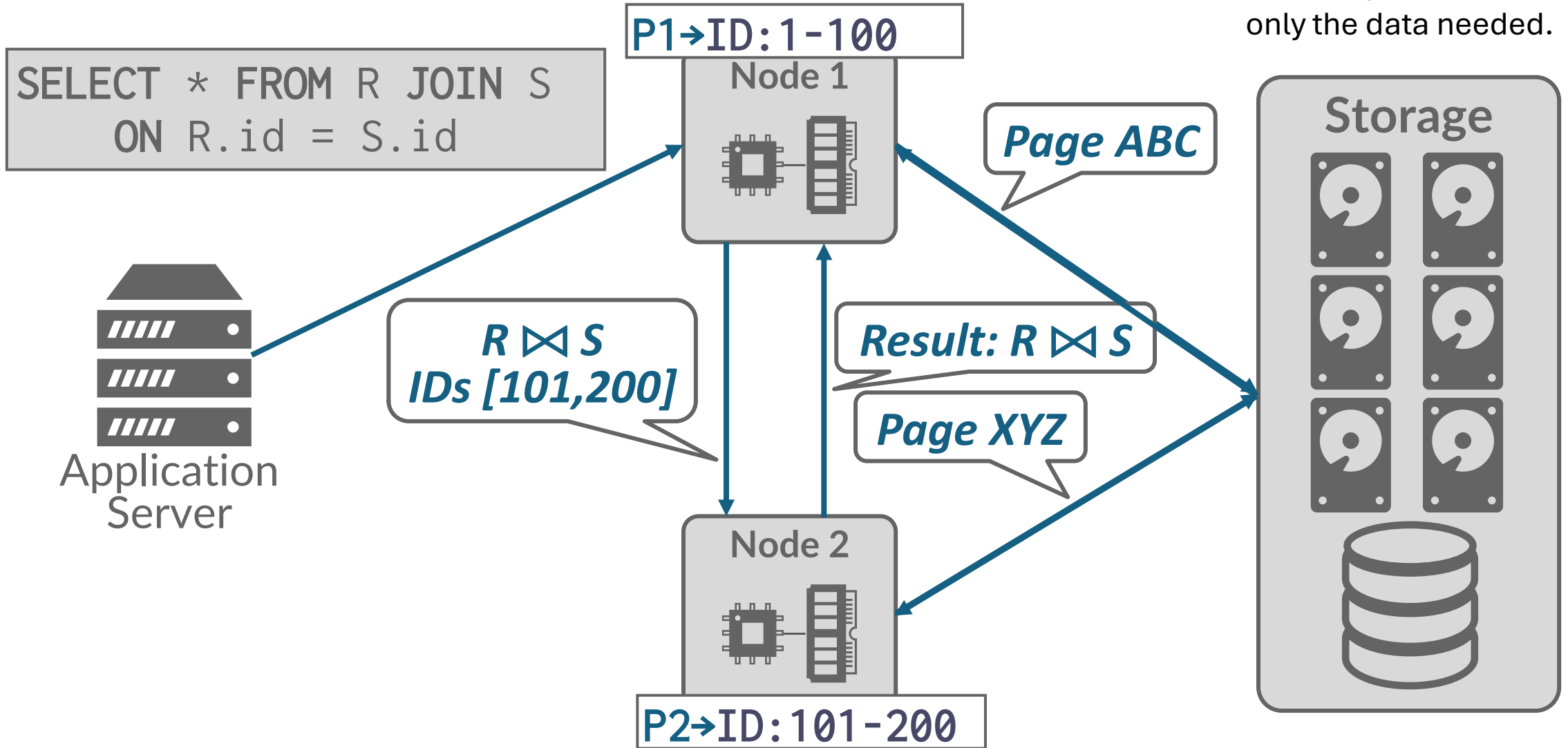
Result: $R \bowtie S$

Node 2



P2→R.id:101-200
P2→S.id:101-200

Pull data to query (shared storage)



Observations

- In many cases, new data arrives periodically and it is common that users are more interested in more recent data than old data.
- Therefore, it is important to use column-oriented storage (DSM) to incrementally store and compress a group of rows rather than all rows.
- A more common format is PAX: hybrid row-column storage
 - Representative formats: Parquet, ORCFile

OLAP summary

- OLAP is a specialization of relational databases to support analytical processing and report generation
 - Typically done in large “batch” operations on the entire database
 - Rule of thumb: pick as “coarse-grained” materialized views or query results as will allow you to construct all the cross-tabs that may be necessary

Cloud databases: outline

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

Credits

- Huanchen Zhang, Tsinghua
- Andy Pavlo, CMU
- Dixin Tang, UT Austin